An Analysis of the Semantic Foundation of KerML and SysML v2

João Paulo A. Almeida[1(✉)], Luís Ferreira Pires[2], Giancarlo Guizzardi[2], and Gerd Wagner[3]

[1] Ontology & Conceptual Modeling Research Group (NEMO), Federal University of Espírito Santo (UFES), Brazil
jpalmeida@ieee.org

[2] Semantics, Cybersecurity and Services (SCS) Group, University of Twente, The Netherlands
{l.ferreirapires,g.guizzardi}@utwente.nl

[3] Institute of Informatics, Brandenburg University of Technology, Cottbus, Germany
wagnerg@b-tu.de

Abstract. In the last decades, Model-Based Systems Engineering (MBSE) has received significant attention, leading to standards such as SysML. SysML is due to a recent and radical update, breaking the dependence of its specification from UML, and leading to the development of two languages: KerML, which provides a top layer of general constructs, and SysML v2, which specializes KerML for systems engineering. In this paper, we analyze the formal and real-world semantics of the proposed KerML and SysML v2 specifications, and draw implications for their improvement and further development. Our attention is focused towards key constructs of the languages. We also examine the approach taken in the specifications to deal with dynamic aspects, which is inspired by the 'four-dimensionalist' view.

Keywords: Model-Based Systems Engineering · Semantics · Foundations · Kernel Modeling Language (KerML) · Systems Modeling Language (SysML) v2

1 Introduction

SysML is a systems modeling language developed in the scope of the Object Management Group (OMG) to facilitate systems engineering. The first major version of the language (SysML v1) [16] was defined as an extension of UML 2 [17] as a profile. This dependence on UML had negative effects on the language because it introduced artificial complexity (for UML compliance). SysML v2 [20] is a major redesign, removing the constraints imposed by UML dependence. SysML v2 also aims to address a significant limitation of SysML v1, namely, that – similarly to UML – it lacks an authoritative commonly accepted formal semantics. This has the consequence that SysML v1 constructs could be interpreted differently by different people [18], including tool builders, posing a threat to model interoperability and reuse.

SysML v2 is expected to play a central role in Model-Based Systems Engineering (MBSE) by serving as a backbone for the development of digital threads that cover

the whole system's lifecycle, from conception (requirements capturing) to operation, possibly including digital twins of the system [22]. In order to fulfill this role, SysML v2 should be able to provide a 'single source of truth' and be based on solid semantic notions, avoiding ambiguity.

In this paper, we investigate how the semantics of SysML v2 is being defined, aiming at assessing its suitability to support model interoperability. Ideally, the semantics definition should enable unambiguous interpretation of SysML v2 models and reliable automated processing. In particular, SysML v2 should be able to support model verification to assess whether the models are valid in accordance with the language rules, model checking to verify properties of the models, simulation, generation of valid instances (universes that are admitted by the models) and possibly code generation in the future.

An integral aspect of the development of SysML v2 is that the language is built as an extension of a more general language called KerML [19]. KerML provides basic (domain-independent) modeling constructs which are reused in the specification of SysML v2. KerML is intended for reuse in other specialized modeling languages in the future, with the ambition to form a family of semantically integrated modeling languages [19]. Hence, the quality of the (semantic) definition of KerML may have consequences reaching beyond SysML v2.

The complexity of the specifications makes our analysis a challenging task. At the time of writing, the main documents specifying KerML [19] and SysML [20] have 407 and 646 pages respectively. The abstract syntax of KerML has 79 classes and that of SysML has 172 classes (based on the present reference implementation in Ecore). Because of this, we focus our attention on key constructs that are required to understand the main design choices in language architecture and semantic foundations. We should also note that the specifications are being finalized at the time of writing, in what OMG calls a Finalization Task Force (FTF). This makes our effort timely, given that there is the possibility of influencing the development of the specifications prior to their main release. Hence, our main contribution is to provide some recommendations and identify alternatives in order to address some issues revealed in the analysis.

The remainder of this paper is structured as follows: Section 2 presents an overview of the KerML/SysML v2 language architecture, introducing the key constructs we analyze; Section 3 discusses how the semantics of these constructs are defined; Section 4 focuses on how dynamic aspects are represented in the languages, revealing implications of the 'four-dimensionalist' approach adopted; Section 5 provides some recommendations based on our analysis; Section 6 draws conclusions and outlines future work.

2 KerML/SysML v2 Architecture

SysML v2 has a complex language architecture. Its abstract syntax is defined in a large monolithic MOF metamodel, and two concrete syntaxes are provided: a textual notation (specified with an EBNF grammar) and an accompanying diagrammatic notation. Furthermore, the specification employs more than one strategy with the intent of providing semantics for its constructs. Given this complexity, we focus our attention here on the overall language architecture and on a couple of key constructs which are required to understand how the semantics of SysML v2 is defined, namely the *Classifier* and *Feature*

metaclasses of KerML and the *Definition* and *Usage* metaclasses of SysML v2[1] itself. Throughout the paper, we strive to remain faithful to the way in which the languages are presented in the specifications (and indicate explicitly otherwise).

2.1 Overview

SysML v2 is specified as an extension of the *Kernel Modeling Language* (KerML [19]). KerML has been developed to be used as a base language for the definition of other modeling languages and has been specified in three layers:

1. the *Root layer*, which defines the "syntactic foundation" of the language: elements, relationships, dependencies, annotations and namespaces;
2. the *Core layer*, which defines the "semantic foundation" of the language: types, classifiers and features (attributes, parts, actions, etc.);
3. the *Kernel layer*, which defines the remaining elements of the language including classes, data types, structures, behaviors, functions.

In addition, KerML defines model libraries with elements that are to be reused across models. Particularly, the *Semantic Library* is meant to give semantics to the KerML metamodel.

SysML v2 reuses the KerML concepts and extends them to enable a more detailed representation of requirements, structure, and behavior of systems, aiming at supporting various system engineering methods and practices [20]. Similarly to KerML, SysML v2 also defines model libraries: (i) the *Systems Library*, which is intended to give semantics to the Systems metamodel, and (ii) the *Domain Library*, which defines fundamental concepts for systems engineering, such as, e.g., geometry, and quantities and units.

Figure 1 shows the SysML v2 language architecture, with MOF (M2) metamodels at the left-hand side defining the abstract syntax of KerML and SysML, and their instantiations (M1 models) at the right-hand side. KerML defines not only the basic elements of the SysML abstract syntax, but also the SysML (formal) semantics, as we discuss in Section 3. In addition to the abstract syntax, both languages are given specialized textual notations, and SysML is also given a diagrammatic notation.

2.2 KerML Classifier and Feature

The KerML Root and Core layers contain the basic language constructs necessary to define KerML models. The MOF language metamodel fragment in Figure 2 shows some of these constructs, starting from `Element`, which represents any element of a model; `Namespace`, which represents containers for elements; and `Type`, which represents types classifying things in the modeled system. `Type` is specialized into `Classifier` and `Feature`, with the latter representing how things are related [19].

Elements in a model may be related through instances of subclasses of the `Relationship` metaclass. We show here only two of those subclasses for the sake of brevity: `Specialization` and `FeatureTyping`. Other relationships (omitted in the

[1] Henceforth, whenever we refer to 'SysML' with no version qualification, we mean SysML v2.

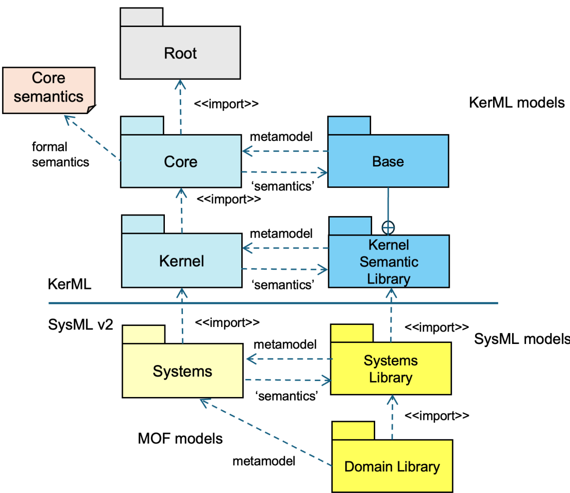

Fig. 1. KerML/SysML v2 language architecture.

diagram) include subsetting and redefinition between features, which are similar to homonymous mechanisms in MOF/UML, as well as disjoining, unioning, intersecting and differencing between types.

`Specialization` relates two types, a *general* one and a *specific* one, requiring "all instances of the specific type to also be instances of the general Type" [19]. `FeatureTyping` is defined as a special kind of `Specialization` in which the specific type is a feature (the `typedFeature`) and, when the general `type` is a classifier, "the `typedFeature` has values that are instances of the Classifier." [19]. (We will turn our attention to this construction later, as we understand it departs from the standard conception for relations and for specialization, making the feature – a relational notion – a specialization of its *co-domain*.)

The KerML specification also defines a concrete textual notation for the language. Listing 1.1 shows a simple example in this notation, defining a classifier `Car` with a feature `driver` typed by `Person`, which relates a car to at most one person as its driver. Feature typing is represented with a colon '`:`', making in this case `Car::driver` specialize its range (`Person`). A feature `mainEngine` typed by `Engine` is specified as a 'composite' feature of `Car`. Finally, a classifier `Sedan` is defined to specialize `Car`.

Listing 1.1. A KerML model specifying persons, cars and engines with Core layer constructs.

```
1  classifier Person;
2  classifier Car {
3      feature driver : Person[0..1];
4      composite feature mainEngine : Engine[1];
5  }
6  classifier Engine;
7  classifier Sedan specializes Car;
```

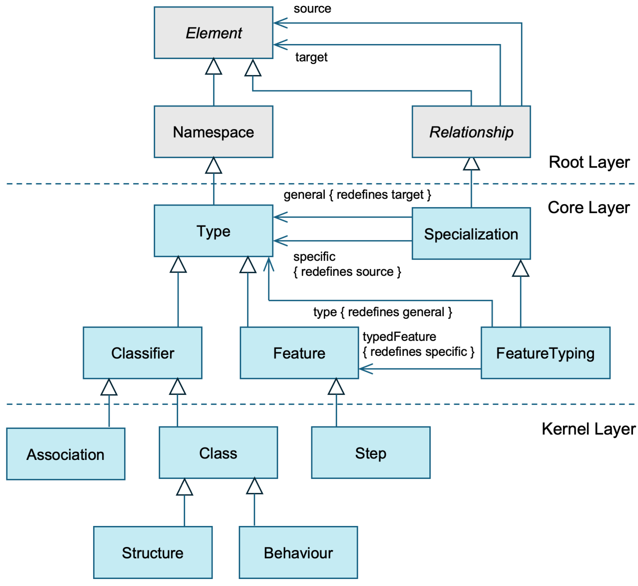

Fig. 2. Some important metaclasses of the KerML MOF metamodel.

The Core layer constructs are specialized in the Kernel layer (shown in the bottom of Figure 2), which introduces the following kinds of classifiers: (i) *Classes*, whose instances are 'occurrences' (a semantic notion subsuming 'objects' and 'performances'), (ii) *Structures*, which are classes whose instances are 'objects', (iii) *Behaviors*, which are classes whose instances are 'performances'. A special kind of feature called *Step* is also introduced to model features which are typed by behaviors.

Listing 1.2 revisits Listing 1.1 using some of these Kernel layer constructs. It includes a behavior `PaintJob` that represents the performances which involve the painting of a particular car, including a `priming step`. The keyword `readonly` is used to indicate the `carToPaint` in a performance instantiating `PaintJob` does not change.

Listing 1.2. A KerML model employing Kernel layer constructs.

```
1  struct Person;
2  struct Car {
3      feature driver : Person[0..1];
4      composite feature mainEngine : Engine[1];
5  }
6  struct Engine;
7  struct Sedan specializes Car;
8  behavior PaintJob {
9      readonly feature carToPaint : Car[1];
10     step priming : PrimerApplication[1];
11 }
12 behavior PrimerApplication;
```

2.3 SysML Definition and Usage

The abstract syntax of SysML has been defined by specializing the KerML metamodel, as shown in Figure 3 with KerML metaclasses in light blue and SysML metaclasses in light yellow. The `Definition` and `Usage` constructs specialize `Classifier` and `Feature`, respectively, inheriting their semantics. `Definition` and `Usage` are further specialized into `OccurrenceDefinition` and `OccurrenceUsage`, further specialized into `ItemDefinition` and `ItemUsage`, which in its turn is further specialized into `PartDefinition` and `PartUsage`. `OccurrenceDefinition` and `OccurrenceUsage` are also specialized into `ActionDefinition` and `ActionUsage`.

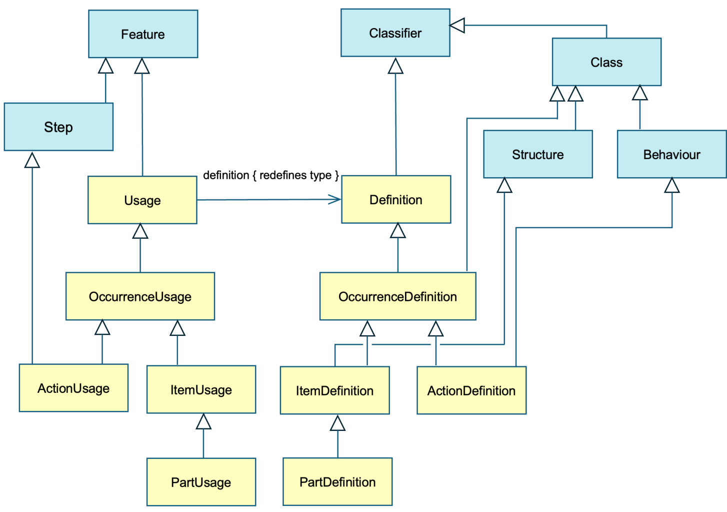

Fig. 3. Specialization of KerML metaclasses for definitions and usages in SysML.

The concrete textual notation for SysML, although bearing some similarity with that of KerML, is fully redefined in the SysML specification. Listing 1.3 revisits the example in Listing 1.1 now with a SysML model. Differently from KerML, SysML also has a diagrammatic notation, but, given our focus on the semantic aspects of the language, we omit this diagrammatic notation in this paper for the sake of brevity. In Listing 1.3, `Person`, `Car`, `Engine` and `Sedan` are specified as *part definitions*. The term 'part' may sound strange when applied to persons and drivers. However, in SysML v2, the term is applied to all modular units of structure (previously called 'blocks' in SysML v1), regardless of whether they play a role in a parthood relation that is explicitly modeled. A car is related to a person through the *part usage* `driver`. Since the person is not a part of the car, it must be modeled as a *referential part usage*, indicated with the keywords `ref part`. The engine is also related to the car through a part usage, but in this case the engine is an actual part of the car and it was therefore described as a *composite part usage*, indicated simply with the keyword `part` (denoting a part usage, by default).

An *action definition* `PaintJob` is introduced with keyword `action def`. It includes a 'parameter' `carToPaint` of type `Car` with direction 'inout' and an *action usage* `priming`, with keyword `action`. Definitions and usages are defined for other concepts such as individuals, ports, connections, interfaces, control nodes, states, transitions, etc., so it is a predominant recurring pattern in the design of SysML.

Listing 1.3. A SysML v2 model specifying persons, cars, engines and paint jobs.

```
1  part def Person;
2  part def Car {
3          ref part driver : Person [0..1];
4          part engine : Engine[1];
5  }
6  part def Engine;
7  part def Sedan specializes Car;
8  action def PaintJob {
9      inout carToPaint : Car;
10     action priming : PrimerApplication[1];
11 }
12 action def PrimerApplication;
```

3 Semantics

A formal semantics, in the spirit of Tarski's model-theoretic semantics for predicate logic, is only defined for the KerML Core layer, discussed below. Furthermore, a model library written in KerML itself (the *Kernel Semantic Library*) is employed to provide semantics to the elements of the KerML Kernel layer. This library is then further specialized in the SysML *Systems Library*, adding system engineering-specific notions.

3.1 KerML Core Layer

The formal semantics for the KerML Core layer refers to a *vocabulary* provided by the names of the *classifiers* and *features* defined as *types* in a KerML model. A feature is informally defined as "a Type that classifies relations between multiple things (in the universe)". According to the specification, "KerML semantics are based on classification: a model has elements that classify things in the modeled system."

An *interpretation* of a vocabulary is a quadruple $\langle \Delta, P, S, I \rangle$ consisting of[2]:

1. a *universe* Δ containing "all actual and potential things the vocabulary could possibly be about" (or, more specifically, data values, objects, links, performances, etc.);
2. a strict partially ordered set P called *marking set*;
3. a set S of *sequences* including all unary sequences (consisting of a single element of the universe) and all alternating sequences like (d_1, p_1, d_2), $(d_1, p_1, d_2, p_2, d_3), \ldots$, where $d_i \in \Delta$ and $p_i \in P$;

[2] We deviate slightly from the specification here in an attempt to clarify the formal model structure, but this deviation is not substantive.

4. an *interpretation function* I that maps each classifier and feature name of the vocabulary to a subset of S.

Consider the KerML model in Listing 1.1 defining the classifiers `Car` and `Person`, as well as the features `driver` and `mainEngine`. Imagine a situation with two persons (John and Mary), two cars (car1 and car2) and two engines (e1 and e1) such that car1 is driven by John, car2 does not have a driver, Mary is not driving any car, car1 has e1 as its main engine, and car2 has e2. Then, sequences like (john), (mary), (car1), (car2), (e1), (e2), (car1, p1, john), (car1, p2, e1), (car2, p3, e2) would be required in set of sequences S for the interpretation of `Car`, `Person`, `driver` and `mainEngine`.

Marking elements (p1, p2, p3, etc.) separate elements of the universe in sequences like (car1, p1, john) and are used to account for features that are 'non-unique' or 'ordered'. For non-unique features, the same pair of elements is separated with different markings, and hence the collection of entities in the feature's range for a particular entity of the domain can be conceived as a bag/multiset. For ordered features, the markings used form a total order, such that the collection of entities in the feature's range for a particular entity of the domain can be conceived as a sequence.

The core semantics provides rules to interpret: the predefined top-level classifier `Anything`, which is present in all models and subsumes all other classifiers in the model, and the predefined top-level feature `things`, also present in all models, subsuming all other features; specialization between types; disjoining between types, when the interpretation of two types has no intersection; feature typing, where a feature is given a 'featuring type' corresponding to the domain, and a 'type', corresponding to the co-domain; feature redefinition, multiplicity, uniqueness, ordering, inverting and chaining.

The semantics is defined in such a way that the interpretation function in our example would map `Car` to {(car1), (car2)}, `driver` to {(car1, p1, john)}, `mainEngine` to {(car1, p2, e1), (car2, p3, e2)} and `Person` to {(john), (mary), (car1, p1, john)} and `Engine` to {(e1), (e2), (car1, p2, e1), (car2, p3, e2)}. Note that all sequences in the interpretation of a feature are also included in the interpretation of the 'type' (or co-domain) of the feature, therefore, all sequences ending with a person, like (car1, p1, john), are included in the interpretation of `Person`, and all those ending with an engine, like {(car1, p2, e1), (car2, p3, e2)}, are included in the interpretation of `Engine`.

Discussion KerML's model-theoretic semantics departs from the standard model-theoretic semantics for structural modeling languages such as RDF(s) [11], OWL [15], and Alloy [12], as well as from formalizations of UML class and ER diagrams in the following way:

1. In addition to a *universe* Δ, interpretations also include a *marking set* and the set of all *sequences* S formed with elements of Δ and of the marking set, to allow for non-unique and ordered features.
2. While in standard model-theoretic semantics unary and binary predicates (such as classes and properties) are interpreted as subsets of Δ and of $\Delta \times \Delta$, in KerML they are interpreted as special subsets of S.
3. Due to its peculiar *FeatureTyping* construct, in KerML, the interpretation of classifiers unusually also contains non-unary sequences.

Concerning the latter point, we believe it results from semantic overload, as the specification seems to conflate in features the notions of: (1) a *relation* between entities (car 'has main engine', 'has wheel'); (2) the *related entity* for a given entity of the domain (a car's 'mainEngine') or *the collection of related entities* (a car's 'wheels', each of which instantiates 'Wheel'); (3) a *function* that produces the related entity or collection; and, (4) the *type that an entity instantiates when in a relation* ('Main Engine', 'Attached Wheel', which in fact specialize 'Engine' and 'Wheel' respectively).

The specification of KerML seems to switch between each of these interpretations indistinctly, for various purposes. For example, to discuss the *specialization of features by other features (Subsetting)*, features are treated as relations (following sense (1) above), with the sequences in the extension of the specializing/subsetting feature included in the extension of the specialized feature.

To discuss the *multiplicity of features*, it becomes necessary to talk about the cardinality of the collection of related entities (in sense (2) above), such as the wheels of a particular car. The same is required to make sense of the *ordering of features*: what is *ordered* is the collection of "values of a feature for a given instance of its domain". We have observed that appeal to the notion of "value" or "values" of a feature is common throughout the specification.

The choice for semantics of features (as specializing classifiers) has the consequence that, while features are defined in the scope of their domain (e.g., `driver` as a feature of `Car` in Listing 1.1), they do not change the extension of their domain (`Car`), but rather the extension of their co-domain (in this example, `Person`). Consider the case of mandatory features (such as `mainEngine` in our example). The presence of this feature requires every car to have a main engine. This effectively changes the intension of `Car`, since now engineless cars are ruled out from the interpretation. However, formally in the proposed semantics, it is the extension of `Engine` that changes as a result, even if it is only optional for an engine to be part of a car.

In our view, it is the type that an entity of the co-domain instantiates when in a relation (in sense (4) above, such as 'Main Engine') that is a *bona fide* specialization of the 'featuring type' (such as 'Engine'). This corresponds to what is termed a 'role' in ontology-driven conceptual modeling [8].

There seems to be another issue with the semantics of features involving *Feature-Typing*, which concerns its interpretation when the general type is itself a feature. How would this interpretation be different from that of a *Subsetting* with the same general and specific features?

While features are given a formal semantics based on tuples, and hence most aligned with sense (1) above, the examples in the specification do not adopt relational terminology which is typically based on verbs (such as 'has driver', 'is married with', 'has engine', etc.), opting consistently instead for nouns written in lower case for features ('driver', 'spouse', 'engine') – suggesting sense (2) discussed above and corresponding to what is called *role name* in UML [17]. In the case of features with multiplicity upper bound greater than one, the specification uses nouns in the plural, e.g., 'wheels' of a car. In this case 'wheels' in the plural would specialize 'Wheel'. This convention does not match the 'is a' verbalization for specializations, i.e., 'wheels is a Wheel' does not parse.

Limitations of the Core semantics The semantics of type union, intersection and difference are not defined at present, although we expect that to be a relatively easy fix. The semantics for ‘abstract’ types (classifiers and features) is not provided either, and would likely require a more involved solution. It is not clear whether coextensional types, i.e., types that have the same extension, are considered identical, as there is no notion of type identity in the specification.

Some properties of features such as ‘directionality’ (in, out, inout) are discussed informally in the text, but not given a formal semantics, and would certainly require more involved structures in the formalization, as they evoke notions of ‘externality’ to an instance. The same can be said of the notion of *conjugation of types*, in which a “type inherits visible and protected memberships from the original type, except the direction of input and output features is reversed”.

The `all` qualifier for types determines that a type includes in its definition not only the necessary conditions for type application (the default interpretation) but also the sufficient conditions; no formal semantics is provided for the corresponding metaproperty of type (`isSufficient`).

Mereological qualifiers for features such as `composite` and `portion` are only defined informally, along with `end` qualifiers and visibility qualifiers (`public`, `private`, `protected`). This means that the semantics is largely underspecified; nothing is present in the formal structure to guarantee the informal interpretation provided for these notions is respected. For example, nothing would prevent an entity from being a `composite` part of itself, or to enforce that “the values of the [composite] Feature cannot exist after its featuring instance no longer does” as required in the specification.

Finally, there is an issue concerning ‘variation’ of the extent of classifiers and features in time. The interpretation function maps a vocabulary element to a fixed set of entities in the universe. Hence, it cannot account for change in the population of classifiers or in features. This makes it impossible to distinguish `readonly` features from features that are not thus qualified (and that could change in time), such as `driver` in our example. Considering the current specification, all possible tuples involving cars and drivers would be in the interpretation of `driver`. This poses a challenge for the semantics of multiplicities for non-read-only features in the present specification.

No validation and verification efforts for the formalization are reported. As far as we are aware, the formalization does not support the verification and validation of properties of models expressed in the language. A machine-readable version of the Core semantics is not made available.

3.2 Kernel Model ‘Semantic’ library

The ‘Core layer’ whose semantics was reviewed in the previous section is specialized by a ‘Kernel layer’. This layer introduces the following metaclasses specializing the (Core layer) `Classifier` metaclass: `Class`, `Datatype`, `Structure`, `Behavior` and `Association`. The specification defines rules for these metaclasses such that their instances in a KerML model are required to specialize certain elements of the Kernel Semantic Library. For example, all *structures* (such as `Car` in Listing 1.2) specialize the `Object` top-level structure in the Kernel Semantic Library, either with explicit specialization declarations or implicitly by default. All *behaviors* (such as `PaintJob`

in Listing 1.2) specialize the Performance top-level behavior in the Kernel Semantic Library. All *datatypes* specialize DataValue, all *classes* specialize Occurrence and all *associations* specialize Link. We omit the treatment of associations further in this paper, noting solely that there is some semantic overlap with features.

The Kernel Semantic Library is written in KerML itself, using also constructs from the Kernel layer. A fragment of the library is shown in Listing 1.4, which omits package structures and features for brevity. The library defines a taxonomy for all entities in the model's semantic domain. The top-level classifier Anything is included explicitly, and is then specialized into Datavalue, Occurrence and Link. Datavalue is the top-level *datatype*, Occurrence the top-level class, and Link the top-level association. Occurrence is specialized into Object and Performance, where performances are disjoint from objects. All of these types are declared abstract, revealing the intention that they should be further specialized in KerML models.

Listing 1.4. A fragment of the Kernel Semantic Library in KerML

```
1  abstract classifier Anything;
2  abstract datatype DataValue specializes Anything;
3  abstract class Occurrence specializes Anything
4                                   disjoint from DataValue;
5  abstract assoc Link specializes Anything;
6  abstract struct Object specializes Occurrence;
7  abstract behavior Performance specializes Occurrence
8                                   disjoint from Object;
```

Most of the content of the Kernel Semantic Library is in the form of taxonomic constraints, more specifically specialization and disjoining declarations as seen in the listing above, as well as feature typing. A few classes have redefinitions for features of their supertypes in order to determine specific admissible values. For example, the *struct* Body is defined as a specialization of Object with feature innerSpaceDimension=3 (a feature defined originally for Occurrence that represents "the number of variables needed to identify space points in this Occurrence" [19]). Some elements in the Kernel Semantic Library are given invariants (presently 46 invariants in total) written in an expression language that is also part of the Kernel layer. Some elements are given so-called 'binding connectors' which "require their source and target features to have the same values on each instance of their domain." This is a simplified form of invariant specification, and there are presently 21 of such invariants in the library.

Limitations of the Kernel Semantics The semantics of the expression language is only defined informally. This means ultimately that the formal semantic content of a KerML model is constricted to: (i) what is represented with the use of constructs that specialize the Core layer fragment as discussed in the previous sub-section (specialization and disjoining between types, feature typing, multiplicity, redefinition, uniqueness, ordering, inverting and chaining), and (ii) what is inherited from the elements of the Kernel Semantic Library that use the formalized Core layer constructs, e.g., that all *structs* are disjoint from *behaviors* that all *behaviors* are disjoint from *datatypes*. A consequence of this is that the semantics of KerML is largely underspecified.

We have observed that the mapping of constructs of the expression language to the formal mathematical structure of the Core is unclear, due to the choices concerning the interpretation function (to have tuples in the interpretation of classifiers). For example, consider the case of an 'extent expression' in the expression language that is introduced with the unary `all` operator that takes a type as argument (as in `all Person`). It is defined as evaluating "to a sequence of all instances of the named type." If we are to follow the formal interpretation, in the case of classifiers, these would include not only sequences of one element (such as (john) and (mary)), but also sequences in the interpretation of features typed by the classifier (such as (car1, p1, john)). This is most likely not the intended semantics for the expression language. Further, what would happen if an extent expression is applied to a feature? What would count as an 'instance' of the feature? According to the formal interpretation function, these would be tuples (or even sequences with markings). We believe informal expressions such as 'instances of a type' in the specification do not correspond to the entities in the co-domain of the corresponding interpretation function. This creates a barrier for the precise interpretation of natural language statements that employ the term 'instances' throughout the text.

3.3 SysML Systems Library

The semantics of KerML is inherited by SysML as its abstract syntax is defined by specializing the KerML MOF metamodel. SysML *definitions* are KerML *classifiers* and SysML *usages* are KerML *features*. The SysML System Library extends the KerML Kernel Semantic Library, with the abstract item definition `Item` specializing Kernel System Library's `Object` and abstract action definition `Action` specializing `Performance`. A fragment of the Systems Library, which is specified in SysML itself is shown in Listing 1.5; again, we omit package structures and features for brevity.

Listing 1.5. A fragment of the System Library in SysML

```
1 // Object, Performance imported from
2 // KerML Kernel Semantic Library
3 abstract item def Item specializes Object;
4 abstract part def Part specializes Item;
5 abstract action def Action specializes Performance;
```

The specification defines rules for the SysML metaclasses, such that instances of `ItemDefinition` specialize `Item`, instances of `PartDefinition` specialize `Part`, and instances of `ActionDefinition` specialize `Action`. The KerML expression language is reused in SysML; in the Systems Library, it is used for a few assertions (5 in total).

4 Time Matters

As discussed above, KerML introduces the notion of *occurrence* in order to generalize both *objects* and *performances*. While we would normally say that *objects exist* and *performances happen*, the specification defines occurrences as "things existing in space and time" [19, 9.2.4.1], and also as anything that "happens over time and space" [19, 9.2.4.2.13]. There is no clear commitment to a particular foundational ontology in the

specification, but the foundational choices can be traced back to the work of some of the contributors of the specification. More specifically, in [1], Bock and Galey (henceforth, B&G) explicitly position their contribution in the camp of "four-dimensional ontology". In this section, we briefly position the four-dimensionalist and three-dimensionalist views in the philosophical literature in order to analyse the foundations of KerML/SysML v2 from the perspective of temporal ontology.

In the philosophical literature, the distinction between objects (also called continuants or endurants), on one hand, and occurrences (also called perdurants, happenings, events), on the other, is a hallmark of what is called a *three-dimensionalist (3D) view*. A 3D framework typically distinguishes *abstract* and *concrete* individuals, with the former being outside space and time, and the latter having a spatiotemporal existence. Concrete individuals are them normally specialized into objects and occurrences. Objects *exist* in time with all their parts and can change qualitatively while maintaining their identity, provided that certain *essential* properties (including essential parts) are preserved. Occurrences, in contrast, *happen* in time, being composed of temporal parts. Objects participate in occurrences, and the latter *are the changes* that happen to these objects. For example, a particular person, say John, can undergo several qualitative changes throughout his life including changes in mass, height, a tragic loss of a limb, etc. However, it is the very same individual that remains the same across these changes.

In contrast, in a *four-dimensionalist (4D)* view (aka'perdurantist' view) [10], one aims at unifying (what we would ordinarily perceive as) objects as well as performances as four-dimensional entities, also termed 'space-time worms' in the philosophical literature. In this view, all concrete individuals can be 'sliced' into 'portions' in time, i.e., they have temporal parts. A defining feature of the four-dimensional approach, as opposed to the three-dimensional approach, is that there is no genuine change in entities: instead, they have temporal parts with different properties. Consider an example in which a car (car_1) is initially painted red. Later, the owner of car_1 decides to paint it black. What is the color of car_1 in the four-dimensional account? In that view, car_1 does not have a single color; instead, its temporal slices, say car_1*BeforePaintJob* and car_1*AfterPaintJob*, which are temporal parts of car_1, exhibit different properties; the color of car_1*BeforePaintJob* is red, and the color of car_1*AfterPaintJob* is black. This leads to a massive multiplication of entities in the face of changes [25]. Furthermore, all occurrences (not only performances) are "frozen" in the time interval they occupy, i.e., they cannot be different from what they are in any aspect (a notion termed *modal fragility*) and, hence, all their parts and all their properties are essential. So, although different temporal slices of the same occurrence can exhibit different properties, the set of time slices composing an occurrence is fixed. This is because occurrences only exist in the past, i.e., technically speaking, there are neither future nor ongoing occurrences [4, 9][3].

Despite being centered around this notion of occurrence, B&G are never clear on what is exactly meant by an occurrence there (and the same applies to the specifications). At first, the reader has the impression that their main interest is on dealing uniformly with entities that exist in time (i.e., a notion of *concrete individual* mentioned above, which is not an exclusive feature of a 4D view). But do they mean by occurrences

[3] An exception comes from very recent work in formal ontology [6] that proposes a special category of occurrences to deal with *ongoing-ness* and possible temporal change of occurrences.

really occurrences in an ontological sense, i.e., perdurants? If so, what exactly is the theory of occurrences adopted? What is the mereological theory adopted for connecting occurrences to their parts? Does their 4D approach depart from a classical theory of occurrences and allow ongoing, future, possibly changing occurrences?

Adopting a truly 4D approach has its ontological and conceptual costs. For starters, there are several mature 3D frameworks [2,8,21], some of which have been successfully employed to ground conceptual modeling languages (e.g., [8]). Additionally, there is evidence that 3D approaches are pragmatically more efficient than their 4D counterparts in supporting the users of models in problem-solving tasks [24]. The authors seem to brush off these difficulties by proposing to hide the ontological complexity of the framework behind a 3D terminology and way of speaking, while claiming that users could be shielded from that complexity by the development of sophisticated tools. It is not presently clear how this can be achieved; for example, how would occurrences in a 4D approach have mutable (non-readonly) features? Even if mutable features were given a formal semantics in the KerML Core, they would remain in principle inconsistent with the 4D approach. The adoption of a 4D approach should be carefully and explicitly motivated, as well as fully developed, as it has far reaching (and often subtle) consequences. For example, the 4D approach taken in the Kernel causes havoc on the Core notion of multiplicity for classes and features. This is because it requires that portions "must be classified the same way as the Occurrences they are portionsOf." Therefore the extent of a *Class* always includes all portions of instances of that Class.

The specification leaves open several questions regarding occurrences and their parts. We will address two of these here. Firstly, the difference between objects and performances is made in the following way: objects are "occurrences that take up a single region of time and space"; performances are "occurrences that can be spread out in disconnected portions of space and time". However, the distinction as such is not adequate, as there are scattered objects (e.g., ranging from bikinis to archipelagos and constellations), which break that rule. Secondly, the authors are not clear about which kind of mereology is adopted for relating occurrences and their parts. Typically, perdurants follow what is called an extensional mereology: a partial-order relation plus the *weak supplementation* axiom (if an occurrence x has a proper part y then there is at least another part z of x that is disjoint from y), as well as the extensionality axiom (two occurrences are the same iff they have the same parts). On top of that, object parthood also requires some modal mereological notions to distinguish between optional parts (e.g., the radio on a car), mandatory parts (e.g., the engine – which is necessary but replaceable by other engines), and essential parts (e.g., the chassis – which is both necessary and irreplaceable). Despite the importance of parts and their composition in systems engineering, it is unclear how these mereological issues would be treated in B&G's framework and in the KerML/SysML v2 specifications.

5 Recommendations

In this section, we provide some recommendations considering the issues we have identified in this paper, covering the language's semantic definition and treatment of

temporal aspects/variation in time. We also identify opportunities to improve on the choice of terminology employed in KerML and SysML v2.

Semantics of the Core layer There are a number of constructs in the Core layer that have not yet been given a formal semantics. So, either a more comprehensive semantic definition or a simplification of the Core layer constructs is required. A more comprehensive semantic definition following the current choice of Core layer constructs seems a daunting task, given that it would require an account of directionality of features (in, out, inout), variation of features in time (readonly versus non-readonly), mereology of features (composite vs. portion features), among others. We believe a simplification of the Core layer would be desirable. A formalization of the semantics of the expression language (currently part of the Kernel layer) would be required to establish a minimal formal basis to be explored further in the semantic libraries.

Formal semantics for features As we have discussed in Section 3.1, there are some negative consequences of the present treatment of features, with tuples included in the interpretation of classifiers. We hold that maintaining the standard separation between monadic and relational types would simplify and clarify the semantic definition. This move would only require to revise the conceptualization of *FeatureTyping* moving away from its current definition as a form of *Specialization*. Whenever a contextualized specialization of a type is required, roles can be employed, in line with ontology-driven conceptual modeling [7]. Alignment with a more standard treatment of relations/properties could contribute to providing a translation of Core layer constructs into existing formal languages for automated processing, such as OWL [15], Alloy [12] and the TPTP First-Order Logics (FOL) syntax [23].

Time matters We would recommend factoring out of the Core layer all considerations of temporal ontology. This would simplify the Core's formal semantics, and, importantly, free language designers reusing KerML from the commitment to a 4D ontology, which has far-reaching consequences for the models produced. Libraries adopting either a 3D or a 4D approach would be able to adopt KerML, contributing to realizing its ambition of forming a family of semantically integrated modeling languages. Also in this case, factoring out of the Core layer considerations of temporal ontology simplifies the formalization, and enables a simpler transformation to languages which do not have a native treatment of time or modality (such as OWL and TPTP FOL).

Terminology matters A revision of terminological choices is also warranted to make the semantics of the various constructs more transparent. Consider the following set of key constructs in the metamodel that are related by specialization: (a) an *ActionUsage* is a *Step* which is a *Feature*; and (b) a *PartDefinition* is a *Structure* which is a *Class*. There are two choices which obscure both taxonomies (a) and (b): first, the shifts between KerML *Classifier* and *Definition* to SysML *Feature* and *Usage*; and, second, the dropping of general *Class* and *Feature* terminology which occurs in the KerML Kernel layer. Using modifiers could improve ease of use of the standard by making sure the inherited semantics of the constructs becomes transparent from their labels. This could lead to revisited taxonomies such as (a) *ActionFeature*, *BehavioralFeature* and

Feature; (b) *PartClass*, *ObjectClass*, *Class*. In this proposal, *ObjectClass* would replace the opaque term *Structure*, appealing to the notion of 'object' which is key to understand the semantics of the construct. Likewise, *PerformanceClass* would clarify the semantics for *Behavior*, also revealing the 'object'–'performance' semantic dichotomy. We observe further that the problems in the taxonomies are compounded with the use of words which have a relational connotation (such as 'part') for non-relational notions. Given the focus of the language on systems modeling, term 'part' could easily be switched for the more accurate term 'system'. So, taxonomy (b) would read *SystemClass*, *ObjectClass*, *Class*. These revised choices consistently improve the similarity between the labels of metaclasses in each taxonomy. We have used LLM embeddings (of OpenAI's `text-embedding-3-large` model) to assess the cosine similarity in the original and in the recommended metaclass taxonomy labels. For taxonomy (a), we have an improvement of similarities from 0.34 ('action usage'–'step') to 0.48 ('action feature'–'behavioral feature'), and from 0.43 ('step'–'feature') to 0.54 ('behavioral feature'–'feature'). For taxonomy (b), we have an improvement of similarities from 0.33 ('part definition'–'structure') to 0.58 ('system class'–'object class'), and from 0.42 ('structure'–'class') to 0.59 ('object class'–'class').

6 Final Considerations

In this paper, we have studied central aspects of the semantic foundation proposed for KerML and SysML v2. We have observed that there are a number of aspects that deserve improvement and proposed recommendations which can be summarized as follows. First, we propose a complete formalization should be provided for the Core layer semantics, preferably in a standard technique; we believe this can be achieved by simplifying the layer's constructs and by revising the formal structure for the interpretation of features. Second, we propose all temporal ontology matters to be factored out of the Core layer; the Core layer then becomes reusable in 3D and 4D approaches. This is particularly important given the conceptual objections to the 4D approach and the empirical evidence of its cognitive cost for users [24]. Third, we propose terminological choices which have the potential to improve semantic transparency of the constructs.

Concerning related work focusing on SysML v2, Jansen et al. [13] have performed an assessment of how the language adheres to some general language engineering guidelines and best practices. They have emphasized the need for SysML v2 to be given a formal semantics, which was not available at the time the paper was published (2022). Given the evolution of the specification, our work complements the analysis they have performed, now with an in-depth perspective on the language's proposed semantic definition. Although there have been a number of efforts to provide a formal semantics for SysML v1 [3, 5, 14], we are unaware of other efforts devoted to the semantics of the current KerML/SysML v2 specifications.

Further work is required in order to study the behavioral aspects of KerML/SysML v2 in detail. This requires delving into the Kernel Semantic Library and into the Systems Library. Given the complexity of these libraries, we believe a reconstruction of key parts of this library in a technique amenable to automated processing is required to make sense of the implications of the library contents.

Acknowledgments. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq (443130/2023-0, 313412/2023-5) and FAPES (2021-GL60J, 2022-NGKM5, 1022/2022). We would like to thank Jim Logan and Ed Seidewitz who provided us with some clarifications during the writing of this paper. Any errors or omissions are ours alone.

References

1. Bock, C., Galey, C.: Integrating four-dimensional ontology and systems requirements modelling. Journal of Engineering Design **30**(10-12), 477–522 (2019). https://doi.org/10.1080/09544828.2019.1642461
2. Borgo, S., et al.: DOLCE: A descriptive ontology for linguistic and cognitive engineering. Applied Ontology **17**(1), 45–69 (Mar 2022). https://doi.org/10.3233/ao-210259
3. Bougacha, R., Laleau, R., Collart-Dutilleul, S., Ayed, R.B.: Extending SysML with refinement and decomposition mechanisms to generate Event-B specifications. In: Theoretical Aspects of Software Engineering. pp. 256–273. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-10363-6_18
4. Diekemper, J.: The existence of the past. Synthese **191**(6), 1085–1104 (2014). https://doi.org/10.1007/s11229-013-0311-3
5. Graves, H., Bijan, Y.: Using formal methods with SysML in aerospace design and engineering. Annals of Mathematics and Artificial Intelligence **63**(1), 53–102 (Sep 2011). https://doi.org/10.1007/s10472-011-9267-5
6. Guarino, N., Guizzardi, G.: Processes as variable embodiments. Synthese **203**(4) (Mar 2024). https://doi.org/10.1007/s11229-024-04505-2
7. Guizzardi, G., Fonseca, C.M., Benevides, A.B., Almeida, J.P.A., Porello, D., Sales, T.P.: Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0. In: Conceptual Modeling - 37th International Conference, ER 2018. Lecture Notes in Computer Science, vol. 11157, pp. 136–150. Springer (2018). https://doi.org/10.1007/978-3-030-00847-5_12
8. Guizzardi, G., Botti Benevides, A., Fonseca, C.M., Porello, D., Almeida, J.P.A., Sales, T.P.: UFO: Unified Foundational Ontology. Applied Ontology **17**(1), 167–210 (2022). https://doi.org/10.3233/AO-210256
9. Guizzardi, G., Guarino, N., Almeida, J.P.A.: Ontological considerations about the representation of events and endurants in business models. In: Business Process Management: 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings 14. pp. 20–36. Springer (2016). https://doi.org/10.1007/978-3-319-45348-4_2
10. Hawley, K.: Temporal Parts. In: Zalta, E.N., Nodelman, U. (eds.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, Spring 2023 edn. (2023), https://plato.stanford.edu/archives/spr2023/entries/temporal-parts/
11. Hayes, P.J., Patel-Schneider, P.F.: RDF 1.1 Semantics W3C Recommendation 25 February 2014 (2014), https://www.w3.org/TR/rdf11-mt/
12. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
13. Jansen, N., Pfeiffer, J., Rumpe, B., Schmalzing, D., Wortmann, A.: The language of SysML v2 under the magnifying glass. J. Object Technol. **21**(3), 3–1 (2022). https://doi.org/10.5381/jot.2022.21.3.a11
14. Lima, L., Miyazawa, A., Cavalcanti, A., Cornélio, M., Iyoda, J., Sampaio, A., Hains, R., Larkham, A., Lewis, V.: An integrated semantics for reasoning about SysML design models using refinement. Software & Systems Modeling **16**(3), 875–902 (Sep 2015). https://doi.org/10.1007/s10270-015-0492-y

15. Motik, B., Patel-Schneider, P.F., Grau, B.C.: OWL 2 Web Ontology Language Direct Semantics (Second Edition) W3C Recommendation 11 December 2012 (2012), https://www.w3.org/TR/owl2-direct-semantics/
16. Object Management Group: OMG Systems Modeling Language (SysML) Version 1.5 (May 2017), https://www.omg.org/spec/SysML/1.5
17. Object Management Group: OMG Unified Modeling Language (OMG UML) Version 2.5.1 (2017), https://www.omg.org/spec/UML/2.5.1, OMG formal/17-12-05
18. Object Management Group: Systems Modeling Language (SysML) v2 Request For Proposal (RFP) (2017), https://www.omg.org/cgi-bin/doc.cgi?ad/2017-12-2, OMG RFP ad/17-12-02
19. Object Management Group: Kernel Modeling Language (KerML) version 1.0 beta 2 (release 2024-03) (February 2024), https://github.com/Systems-Modeling/SysML-v2-Release
20. Object Management Group: OMG Systems Modeling Language (SysML) Version 2.0 Beta 2 (Release 2024-03) (February 2024), https://github.com/Systems-Modeling/SysML-v2-Release
21. Otte, J.N., Beverley, J., Ruttenberg, A.: BFO: Basic Formal Ontology. Applied Ontology **17**(1), 17–43 (Mar 2022). https://doi.org/10.3233/ao-220262
22. Pessoa, M.V.P., Ferreira Pires, L., Moreira, J.L.R., Wu, C.: Model-Based Digital Threads for Socio-Technical Systems, p. 27–52. Springer International Publishing (2022). https://doi.org/10.1007/978-3-030-97516-6_2
23. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017). https://doi.org/10.1007/s10817-017-9407-7
24. Verdonck, M., Gailly, F., de Cesare, S.: Comprehending 3D and 4D ontology-driven conceptual models: An empirical study. Information Systems **93**, 101568 (2020), https://www.sciencedirect.com/science/article/pii/S0306437920300582
25. Zamborlini, V., Guizzardi, G.: On the representation of temporally changing information in OWL. In: Workshops Proceedings of the 14th IEEE International Enterprise Distributed Object Computing Conference, EDOCW 2010, Vitória, Brazil, 25-29 October 2010. pp. 283–292. IEEE Computer Society (2010). https://doi.org/10.1109/EDOCW.2010.50