[BR23] M. Broy, B. Rumpe:
Development Use Cases for Semantics-Driven Modeling Languages.
In: Communications of the ACM, Volume 66(5), pp. 62-71, ACM, Mai 2023.
www.se-rwth.de/publications/

DOI:10.1145/3569927

**Choosing underlying semantic theories and definition techniques must closely follow intended use cases for the modeling language.**

BY MANFRED BROY AND BERNHARD RUMPE

# Development Use Cases for Semantics-Driven Modeling Languages

DEVELOPING SOFTWARE AND software-intensive systems always requires the development of models. Some are formulated explicitly in some kind of modeling language. Others exist only as mental models and are finally represented implicitly by programs. In fact, application software realizes specific models that are available in the application domain and additionally includes specific technical implementation concepts in terms of hardware, operating systems, and further elements of the required software stack. In the end, application software appears to be a complex, entangled mixture of application models and implementation technology. Like programming languages, Modeling languages are defined by their syntax, which describes the form of the language constructs—textual or graphical—and by their semantics, which specifies their meaning. We claim that the choice of the underlying semantic theories and definition techniques must closely follow the intended use cases for the modeling language. The choice of the syntax should be guided by the semantic domain and its underlying theories and not the other way around.

Modeling languages, such as UML, SysML, and various domain-specific modeling languages, have been suggested to support the specification and construction of systems for specific domains. This leads to the following key questions:

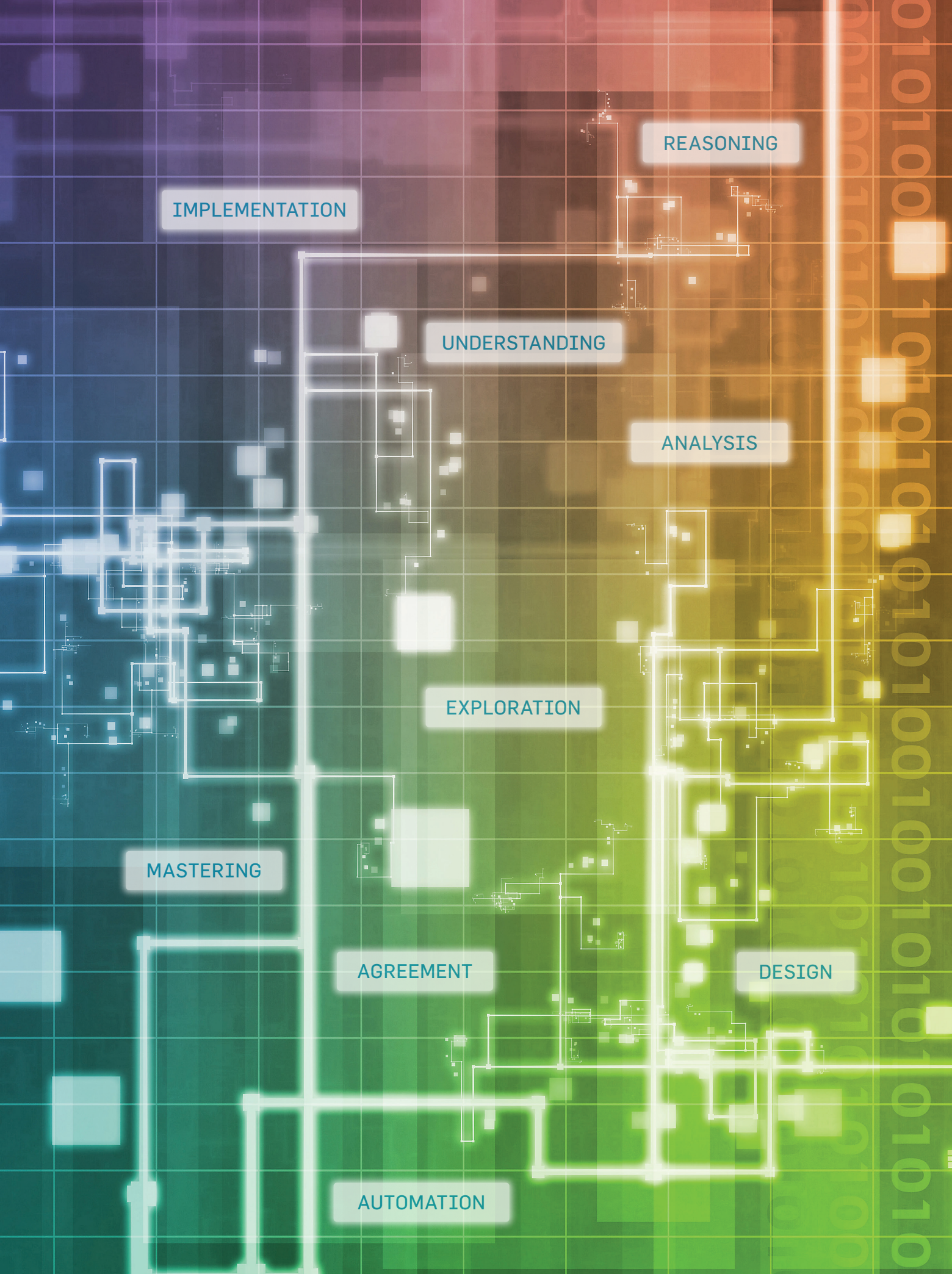
- How should we model real-world entities of systems and how can we relate these entities to their models precisely?
- How should we link up the syntactic representations of models in a modeling language with their semantics?
- How can we represent, specify, and use semantics?

Two questions that arise here are how can a modeling language, a modeling method, and a modeling theory be used systematically, and which aspects and steps of system modeling and system development do they support? Investigations, for example on model-driven engineering,[24] SysML,[25] or variability

## key insights

- **Designing applications in software and software-intensive systems requires the creation, explicitly or implicitly, of models. The more explicitly and deliberately this is done the better.**
- **Professional modeling requires explicit modeling languages—for example, UML, SysML, or domain-specific languages.**
- **Semantics is needed to give meaning to syntactic constructs in these languages. For optimal utilization, the choice of underlying semantic theories and definition techniques must closely support the intended use cases for the modeling language.**
- **The design of modeling languages must be guided by the semantic concepts and their underlying theories and not by pure syntactical considerations.**

IMAGE BY KENTOH


REASONING
IMPLEMENTATION
UNDERSTANDING
ANALYSIS
EXPLORATION
MASTERING
AGREEMENT
DESIGN
AUTOMATION

in modeling language semantics,[6] partially address these issues, although not generally from a usage point of view for models and modeling. Model analysis in various forms and code synthesis from models have already been investigated in early influential modeling languages, such as ROOM[22] or Statecharts.[12]

### Modeling in System Design: Use Cases for Modeling Languages

In the following, we consider modeling languages that are mainly used for modeling software-intensive systems, their structure, their interactions, and their behavior. We study the systematic use of modeling languages and their underlying theories, as for instance demanded in Harel and Rumpe,[13] within the task of system development.

What modeling languages are good for and how they should be used may appear to be rhetorical questions. Of course, modeling languages are intended for describing relevant aspects of systems. In fact, there are quite different use cases for models.

Currently, developing modeling languages involves quite a lot of effort. This is the case, for instance, with SysML in its significantly extended Version 2.0.[10] Too often, however, there are only vague, incomplete, or largely diverting visions of how a modeling language should be used. For SysML 2.0, at least, there is hope: Although there is currently no document available that explicitly describes the intended use cases of SysML models, there seems to be some agreement among the standardization stakeholders on the use cases at least in highly relevant areas. We hope that the emerging SysML 2.0 standard will be accompanied by an explicit list of use cases for the language and therefore definitions of purpose.

**When we are modeling systems, one very significant decision is which aspects of systems are to be modeled and which modeling concepts are chosen.**

The Request for Proposal for SysML 2.0, however, and especially its Section 6, contains a lot of wishes with regard to the SysML 2.0 language syntax. These wishes are rather detailed. They can also be understood as a solution description, but they can no longer be understood as requirements that explain the need for the constructs and specific forms of use of the language. These requirements mainly describe what modeling concepts the language should include. They describe neither what the language should be able to express (even though this can be imagined) nor the kind of techniques, such as model checking or logical verification, the language models should support. Such a list of applicable techniques is especially relevant because it enforces some restrictions in modeling power, as we know from formal methods and corresponding semantic theories, such as algebraic specifications;[11] system description calculi, such as Hoare's Communicating Sequential Processes; the Focus approach for asynchronously communicating distributed components;[4] and various other forms of logic calculi.

Clearly, one obvious purpose for modeling languages is to offer a formalized syntax for creating tools that support model-driven systems engineering (MDSE). Then, not only do developers use models in the development process, but the process is structured and guided based on models.[23] However, a closer look indicates that there are radically different ways in which semantic theories, methodologies, and tools may support MDSE. Table 1 shows a rough classification of use cases for modeling during development.

The availability of fast compilers, sophisticated development environments,

## Glossary

**A *model* is a reduced, respectively abstracted representation of an original in terms of size, detail, and/or functionality and has a purpose with respect to this original. (Entity/Relationship diagrams or Statecharts are examples of models).**

**A *system modeling language* can be understood by humans, processed by computers, and is used for modeling all relevant aspects of and about the systems under development or examination. (UML and SysML are examples of system modeling languages.)**

**A *theory* is an analytical tool for understanding, explaining, and making predictions about a given subject matter—for example, automata provide such a theory.**

***Semantics* describes the meaning of a model. For example, Statechart models get semantics using automata theory. However, an alternative viewpoint is that automata theory is made practical through a concrete modeling language, namely the Statecharts.**

**A model is *valid* if its semantics correctly describes the original.**

or frameworks to automatically execute tests introduced agile development methods as well as entirely new use cases for models. The high degree of efficient automation available enables efficient and early feedback to the modelers. Agile development can be perfectly applied to various kinds of system modeling languages if the languages have precise semantics. Use cases for models may include assistance for sophisticated automatic checks, such as type checking, flow analysis, or even model checking on probabilistic models.[16] It is only when using abstract, high-level modeling languages that the designs remain explicit and can be smartly analyzed. For example, a state machine can easily be analyzed for completeness. If it is encoded as a state design pattern, a code analyzer cannot normally recognize the developer's mental model anymore. To embrace evolution, refactoring[8] can easily be applied to explicit structural and behavioral diagrams. Modeling enables refactoring and thus evolution[19] or calculation of semantic differences[5,17] between the models. Continuous integration is a technique that fits particularly well with MBSE.

Table 1 also describes additional use cases for modeling during system operation, leveraging the close connection between digital twins and models as used in systems engineering and thus assisting DevOps approaches.

A further, highly relevant question concerns the aspects of systems that are modeled: system structure, behavior, or both; interactions between system and environment or internal interactions; data structures; or even quality properties. In the end, a key question is how behavior is modeled. First, there are radically different mathematical models of system behavior at different levels of abstraction. Moreover, the description of behavior may be operational (defining a form of execution), or descriptive (capturing properties with usually many possible realizations).

Clearly, the use cases envisaged should largely determine the design of modeling languages. A language for architecture design looks different than one for simulation or implementation, or one for property specification. Therefore, a careful analysis of use cases is needed as a basis for designing a modeling language.

**Table 1. Classification of needs for semantic-driven models.**

| Needs for modeling during development of systems and software | |
|---|---|
| M1. Specifying systems in requirements engineering | M2. Formulating and evaluating design alternatives |
| M3. Describing system aspects or views for communication | M4. Designing system architectures |
| M5. Describing systems for validating desired system properties in simulations | M6. Collecting user feedback through visual simulations, prototypes, and mock-ups |
| M7. Modeling variants in product lines | M8. Defining reference models for the capture, design, or implementation of requirements |
| M9. Statically analyzing or verifying design decisions | M10. Efficiently evolving designs |
| M11. Understanding semantic differences between versions | M12. Describing detailed system behavior for generating software parts |
| M13. Implementing/realizing/synthesizing systems in general | |

| Needs for modeling during system operation | |
|---|---|
| M14. Customizing systems | M15. Monitoring running systems |
| M16. Capturing deviations between desired or even optimal (modeled) properties and observable (realized) system functionalities | M17. Documenting systems |
| M18. Capturing system execution traces and labeling them with model elements, thus linking system and traces reliably | M19. Optimizing systems |

Modeling (and programming) languages are defined by their syntax ("what we see") and their semantics ("what it means"). Syntax shapes visual representation and semantics defines the meaning of the syntactic constructs. Useful semantic descriptions of meanings contribute to the understanding and the use of languages. A semantic definition is embedded in a helpful theory and a methodological framework that ideally offers a rich set of insights which contribute to the understanding of what the language is modeling as well as techniques (tool-assisted) for deriving properties. This includes questions of specification and verification, for making information and relations explicit or even for synthesizing executables.

## Modeling at Work

When we are modeling systems, one very significant decision is which aspects of systems are to be modeled and which modeling concepts are chosen. We actually need a deep understanding of all aspects that are relevant for the systems to be represented in a model. This serves as a guide for designing modeling languages.[9,15]

A classical approach to designing systems aims at the following aspects (disregarding pure physical aspects for now):

- The *context* of a system, comprising:
  - Neighboring systems
  - Environmental facets and effects
  - Users operating via the human-machine interfaces
- The *system boundary* that determines the system interfaces
- The *system interfaces*, including interface signature (or interface types, the *syntactic interface*, consisting, for instance, of formal parameters or ports), and *interface behavior*
- The system's external and internal structure (also called external and internal system architecture), such as:
  - The functional architecture, a structured model that describes the system's functionality
  - The subsystem structure that addresses the component architecture, the structuring of a system into several layers of subsystems down to reusable and predefined components
  - The interaction of composed subsystems to define the overall system
  - The data structure (also known as the data architecture) of the system that describes how to internally store and manage data.

In addition, we need the following:

- A *quality specification*
- A *technical architecture*. For information-processing subsystems, this consists of hardware and software

models, a software architecture mapped onto virtual and real CPUs, scheduling issues, and operating and bus systems. In addition, the technical architecture describes peripheral devices, such as sensors, actuators, and interfaces to neighboring systems as well as user interfaces.

Most of these aspects can be modeled in logically precise form. Moreover, probabilistic aspects or approximation techniques are often relevant as well. In fact, it is essential that unknown or undecided properties can be treated in modeling by means of sufficiently precise underspecification as a general form of abstraction.

When designing a modeling language, we have to represent the abovementioned concepts with syntactic constructs. For instance, interfaces are described by syntactic modeling elements, such as ports or messages, and their types as parts of interface signatures.

## Selecting Syntax

It is mandatory to define a concrete representation for the concepts and aspects addressed by the language syntax as a set of language constructs and their use in the language structure. First, we introduce several rules, called context conditions, to define and describe a syntactically well-formed model. These ensure that, for instance, identifiers are introduced before they are used and have no name clashes or inconsistencies. Typically, this includes an elaborated-type system and many other language-specific forms of consistency checks.

For programming and other textual languages, a context-free grammar is appropriate. Grammars can also be used to construct an abstract syntax representation without any syntactic sugar. A similar approach applies for diagrammatic languages, where the core language is defined by "meta-models," mechanisms similar to class diagrams. Figure 1 shows, however, that more syntactic representations are possible. In all cases, additional rules are based on the abstract syntax to define context conditions for well-formedness.[7] Well-formed syntactic representations are the basis for defining the semantics of a language.

**Figure 1. Four syntactic representations of a state machine: diagram, math, textual, and tabular.**

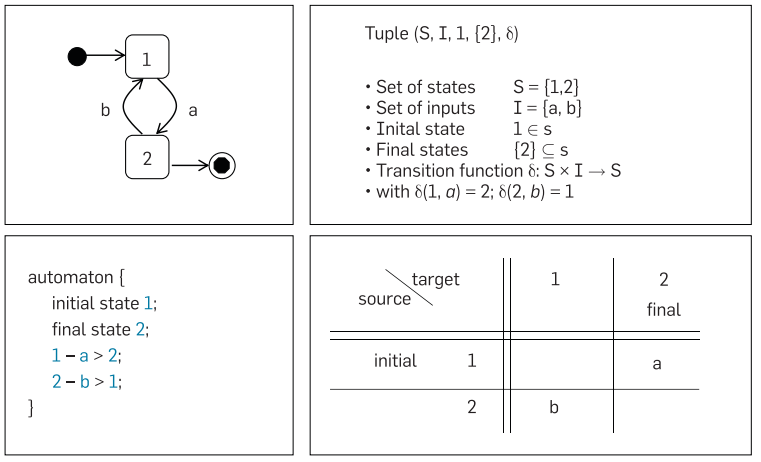



**Table 2. Use cases for language semantics.**

| Use cases for the semantics for language design | |
|---|---|
| UC1. Understanding the meaning of languages. | UC2. Abstraction of the syntactic representation. |
| UC3. Standardization of meaning. | UC4. Standardization and guideline for implementation. |
| UC5. System specification and verification. | |

| Use cases for the semantics when modeling | |
|---|---|
| UC6. Refactoring, that is, ensuring semantically equivalent transformation of models. | UC7. High-level analysis of models. |
| UC8. Proving properties of models and the systems they describe. | UC9. Correct synthesis of optimized realizations. |

## Defining Semantics Formally

Semantics gives meaning to syntactic constructs of modeling languages so that they do not introduce meaningless or invalid models. The key motivation for giving meaning to formal languages is to provide a common understanding of what is expressed by the sentences—that is, the syntactic expressions of a language. Over the last 60 years, a rich collection of different ways of defining the formal meaning of programming languages has been created. The techniques for precisely describing the semantics of modeling languages are motivated by their intended use cases, which we classify in Table 2.

As a relevant side effect, a standardized meaning is a prerequisite for standardized tools. It reduces or even prevents vendor lock-in and enables much better tool interoperability in tool chains, while semantics in and of itself is an enabler for a sophisticated tooling infrastructure.

For a deeper understanding, the following is a list of the most important branches of the various semantics definition methods for the semantics of programming languages. The list also explains how the methods contribute to different ways of using a language:

▸ **Operational semantics.** Mainly given in terms of an evaluation calculus, such as SOS by Plotkin,[20] which defines an idea of evaluation and a guideline for implementation, and if done properly, could even be used to generate an interpreter.

▸ **Denotational semantics.** Here, some type of explicitly defined semantic domain is used and the entities of the formal languages are mapped onto this domain, thus representing the meaning in a mathematically precise way. Domain theory was originally developed by Scott and Strachey.[21]

▸ **Algebraic and transformational semantics.** Algebraic semantics and transformational semantics are inspired by the way math transforms and solves equations. They introduce equivalences and congruencies be-

tween the elements of the formal languages such that syntactically different models that have the same meaning (see Broy et al.[3]) and can be replaced are explicitly defined. Using equivalences as transformations allows us to rewrite, refactor, and restructure models into more adequate forms.

▸ **Logic-based semantics.** Logic-based semantics either gives rules for translating syntactic terms of the language into logical formulas or adds rules of a logical calculus into the modeling language. This aims at specification, verification, and refinement formalisms using logic formulas and calculi. For instance, the Hoare Calculus[14] can also be seen as a syntactic extension of a programming language by assertions. This approach shares characteristics of denotational semantics (mapping to logical denotations) and algebraic semantics (defined by calculi of transformation rules).

A helpful semantic definition does not merely formalize what is expressed by a modeling language; it provides insight and a methodological background and deduction rules. This can be achieved by embedding the language into a logical calculus and user-friendly visualizations of derived results. This is helpful for programming languages but even more so for systems modeling languages.

As for programming languages, modularity by encapsulation and information hiding is related to the idea of abstraction and reusability. Ideally, meaning is given, as in denotational semantics, not just for complete models, but also for their parts, such that the meaning of a composed model is derived from the meaning of its parts. This modularity is especially useful if the composition structure, as described by the architecture of the system, is congruent to the composition structure of the defining models. Then, system interfaces correspond to model interfaces, and modular subsystems and components correspond to encapsulated "sub-models", and so on. As a prerequisite, we need to know when sub-models are coherent and can be composed to describe the desired system.

**The techniques for precisely describing the semantics of modeling languages are motivated by their intended use cases.**

## Abstraction in Modeling Languages

We have seen that there is a large variety of approaches for defining the semantics of programming languages. For modeling languages, the situation is even more complex, because modeling intrinsically relies on additional forms of abstractions. A powerful concept for abstraction is underspecification, allowing sets of possible realizations to be described in one model. Underspecification in systems (as well as software) modeling languages is a concept for tackling at least four important goals:

1. Underspecified models capture the aspects the developer is currently dealing with and abstract away from yet unknown technical details, implementation issues, irrelevant parts, and so on.

2. Flexible handling of design decisions allows developers to avoid premature overspecification by not enforcing arbitrary choices. Here, techniques such as underspecification help to specify bandwidths for relevant system properties without fully determining the resulting system properties yet.

3. Self-adaptive, autonomous, and other smart systems exhibit behavior depending on internal states accumulated over their history or prior training. Their specifications must describe the bandwidth of adaptable behavior requiring abstraction and underspecification as an intrinsic modeling technique.

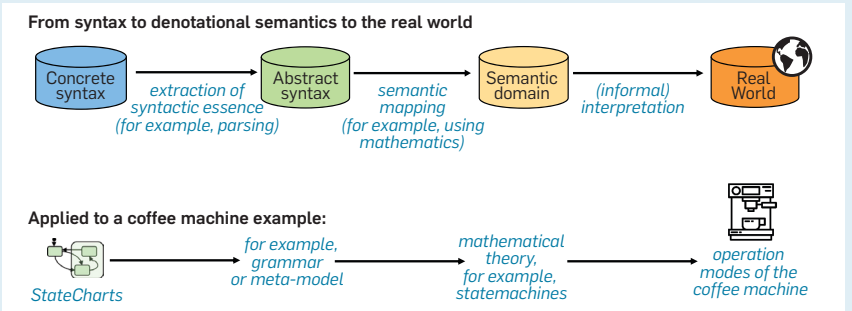
4. In product lines, the common structure and behavior is by nature underspecified. Explicit underspecification techniques also allow us to capture common properties of alternative features and constrain future and yet unknown feature realizations.

Programming languages are typically deterministic, and their programs are thus specified in all details, while underspecification and thus not fully determined models are intrinsic in modeling. Both programming and modeling languages, however, share the fact that precisely defined semantics allow for reasoning about the models and the properties they imply, even though during design, a model cannot and should not capture all technical details. We distinguish clearly between the precision of a model, that is, its degree of detail, and the precision of the semantics defined for the modeling language. Furthermore, if reasoning is to be done in a formal manner, then the semantic description must be embedded into an appropriate logical calculus, allowing automated deductive analyses and constructive syntheses.

**Table 3. Use cases for applying modeling languages in development and operations.**

| |
|---|
| D1. Specify a system to agree in a team on a specific system behavior. |
| D2. Describe the interface of a system/component. |
| D3. Describe properties for a component without looking at its internal structure. |
| D4. Describe the interplay of components as composed in a modeled architecture. |
| D5. Given a model, refine it such that additional properties hold. |
| D6. Given an interface model and the behavioral description, find a decomposition that implements the behavioral description, so that the decomposition is actually a refinement. |
| D7. Given two models, check whether they are equivalent and if not, find the inhibiting model elements. |
| D8. Given two models, check whether one is a refinement of the other. |
| D9. Given a test (for instance, input/output sequences and a property specification), analyze, simulate, or execute the model in such a way that it results in a test verdict. |
| D10. Check whether a model guarantees specific properties. |
| **During system operation there are more use cases for models, such as:** |
| OP1. Use variability, underspecification, and parameterization in the models to optimize real system behavior. |
| OP2. Use models of a system as meta information for sensor data—for instance, describing the data source. |
| OP3. Measure deviation between modeled and observed behavior according to some appropriate metric models |

**Figure 2. Mapping syntax to semantics to the real world.**




For a scalable modeling technique, it is most important that composition is compatible with underspecification. That means the underlying theory must provide composition and refinement techniques in such a form that refined specifications of components by construction lead to refinement of the composite, such as in the Focus approach.[4]

## Model-Specific Semantics Adaptation

A current trend is to define individual semantics for each given syntactic model instance instead of a generally defined and agreed-upon semantics for the modeling language holistically. At first glance, this looks like a linguistic faux pas. What does it mean to use some syntax without standardized semantics? However, one way to accept such an approach would be to understand this as semantic adaptation for a given syntactic language—that is, as a syntactic and semantic framework which allows for adaptation on both sides. The more coherent and precise the semantic framework is, the clearer its adaptability options are, but also the easier it is to reuse and exchange models between different tool chains or projects. However, it does not seem to be a good idea to start with syntax only.

A common semantic framework allows us, for example, to capture a shared understanding of agreed-upon syntactic concepts as well as understand the differences between individual semantic interpretations for a syntactic formalism (as explored for Statecharts in Beeck,[1] for instance), where the challenge is mainly to select the most appropriate interpretation. Associations in class diagrams, actions in activity diagrams, and many other language constructs unfortunately share similar variances.

While semantic adaptations have the advantage that we could give each language model an individual meaning, this also has severe drawbacks. Each syntactic model potentially has a different meaning, which makes syntactic models ambiguous and less useful.[13] Developers have the additional burden of defining semantics, and tool interoperability gets lost.

To some extent, this trend combines the semantics of the language in terms of meaning, for example defined in denotational semantics, and a concrete execution of an implementation of a concrete model in a software stack. Only the latter is dependent on an underlying technology and may need individual technical but not semantic adaptations.

Another interpretation of this trend would be that describing semantic adaptations for model instances does not mean you are defining semantics for the language—it is merely an additional syntactic description added using a language extension. This would be similar to Hoare logic assertions as part of a program or also stereotypes in UML and SysML. In these cases, the languages are extended syntactically, which also requires semantics.

There are approaches for using structural modeling techniques, such as an ontology, a class diagram, or even a SysML architecture (IBD/BDD) to model behavior and thus also time. Encoding a state machine into a class structure is a perfect design pattern but it prevents sophisticated tooling for analyzing the models. It normally also leads to a loss of expressiveness, borne out in many of the UML semantics definitions that used mappings to semantically insufficiently expressive formalisms prominent around the year 2000. The semantic domain and the formalism representing it must be carefully defined and should be free of expressivity shortcomings. Class diagrams are simply not well suited for describing the specific nature of time and behavioral progress and the clear distinction between past, present, and future.

An ontology of the system domain is a good starting point for designing an appropriate modeling language for the domain. Concepts and their relationships, as defined in ontologies (see Mayr and Thalheim[18] for a critical assessment) cannot, however, adequately describe the behavior of systems, their interactions, dynamic changes, and so on, and thus cannot replace language semantics.

Furthermore, an ontology language is also a modeling language and not a

semantics definition per se. However, modeling languages such as SysML have a broader set of language constructs and use cases. Consequently, a useful approach is to embed an ontology of the system domain in an ordinary system modeling language—for example, as reusable modeling elements defined in a library.

### Use Cases for Applying Modeling Languages

To create a useful modeling language and to evaluate whether it serves specific purposes, use cases or even illustrative user stories are indispensable. This is like the creation of requirements in software design and indicates that the development of a modeling language shares similarities with the development of software. However, the "user stories" could alternatively be called "modeler stories" or "developer stories" that address the methodological steps in development.

Developer stories describe how the modeling approach should be used in terms of system development steps and which aspects of system development the approach is intended to support. This is then also related to tooling concepts for that modeling language.

We do not give comprehensive descriptions of the idea of developer stories but in Table 3, we refine the initial list of needs of Table 1 into a list of examples of how modeling languages may be used during system development and operation.

Developer stories and use cases help us to understand the intended use of a model-based system development approach. They show which models and sub-models are required, which modeling aspects are aimed at, and which concepts offer support in certain modeling steps. Developer stories for modeling approaches are very helpful because modeling languages and sophisticated tooling are usually complex.

### Relating Models to the Real World

Having formulated models in a modeling language with a given semantics, a key question is how these models relate to real-world entities that they should model—that is, in the case of system modeling, to real systems. This completes the mapping from language syntax to the systems, as shown in Figure 2.

To a large extent, linking relationships between models and the real world is determined by the concepts that have been identified for and provided by the modeling language and for the entities of the real world to be studied. Given models based on appropriate semantic theories, which operate on the semantic domain, we can run analyses or perform well-chosen experiments leading to distinguished observations related to properties, as represented by property models. Note that the notion of property is available as a semantic concept as a "property of a system," and it is also available as a syntactic concept as a "property model" described in a property-modeling language. A useful modeling language must therefore include such a property-modeling language so that we can define properties and determine whether the properties implied by the model also hold for the real system as justified by observations of this system.

Choosing the syntax of the modeling language appropriately and the carefully basing the semantics on suitable modeling theories allows a direct comparison of deduced properties of the model to the properties observed for real systems. This is a major goal of modeling—models are useful when they allow us to reason about the real world. Semantics, therefore, must support the derivation of properties from system models. Therefore, semantics must be chosen carefully to ensure that derived properties from a correct model also hold for real systems.

What properties can be captured and derived depends on the expressive power of the modeling language and a calculus to derive propositions. Models need expressive power to describe relevant system properties, and there must be a clear definition and understanding of how the model captures properties related to the real system.

### Semantics Targeting Use Cases for Modeling

As explained, the intended use cases for modeling languages guide how the semantics of the language is chosen. Today, it still too often seems to be the other way around: Syntax is defined first and then various discussions about semantics ensue. At the end, there is no clear idea of how the modeling approach should be used, which system elements (components, events, processes, and so on) it shall specify, and what the syntax formally expresses.

We see understanding, reasoning, analyzing, justifying, agreeing, designing, exploring, mastering complexity, implementing, and automation during development as general goals when using a modeling approach in development.

These main goals are not without mutual conflicts. We are also tempted to add explainability and rationale definitions as additional goals, meaning that a running autonomic and self-adaptive system can explain its concrete operative decisions with respect to the original development models. Consequently, a modeling approach must be evaluated according to the key properties shown in Figure 3 to support reflection and communication among developers.

Ultimately, the construction of models has two main purposes that address separate issues. On one hand, prescriptive models denote a starting point or a step in the design of systems where they help to capture properties of

**Figure 3. Key aspects of a modeling approach and their relationships.**

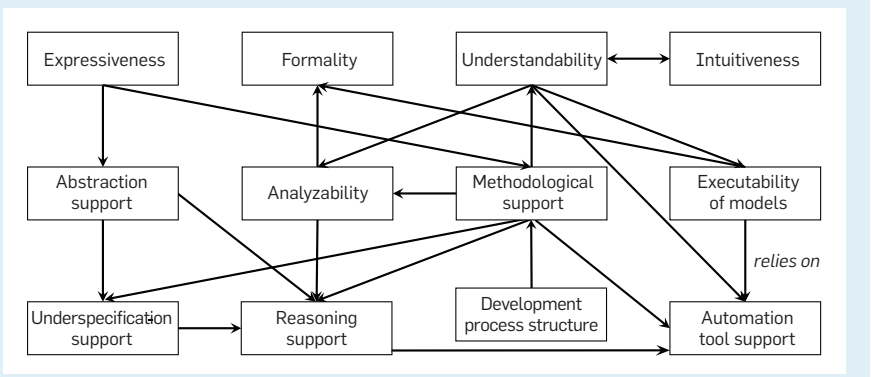
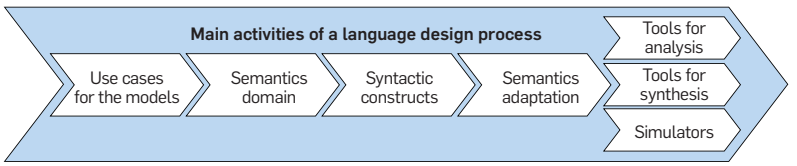


**Figure 4. Main activities of a language design process.**




the systems under construction as a guideline for design and a goal for correctness. On the other hand, descriptive models are used to describe real systems at appropriate levels of abstraction. They can then be used to study properties of these systems, perhaps also to simulate them, to obtain a basis for analysis and observation validation.

These two different forms of support are of course connected. On one hand, when using models for design, we hopefully end up with the creation of a real system. We may then ask whether the model sufficiently describes the real system that is ultimately constructed. On the other hand, we can analyze whether the system constructed is correct for the properties expressed by the model.

Ultimately, there are two areas of use for modeling: expressing ideas about systems to be constructed and collecting properties of systems that are to be analyzed. Both are related; however, different modeling techniques may be more appropriate to address one aspect or the other aspect because, at the very least, tooling for both is quite different.

In practice, a model typically offers many different use cases for different stakeholders in the development process. Some developers constructively design systems in term of models, others make performance optimizations, and others are concerned with security assessments or safety aspects. These different use cases lead to trade-offs in the required detail of models, levels of abstraction, degree of underspecification, precision, and so on, which must be handled by appropriate language constructs and methodology. While the definition of "model" states that there is a purpose with respect to the system, in practice, a model has many purposes and different use cases for different stakeholders, so the corresponding modeling language serves many purposes as well.

As previously mentioned, there are many different approaches for describing the semantics of programming languages and modeling languages. The approach can be denotational, operational, axiomatic, transformational, or logic-based depending on how the semantics are intended to support specific use cases.

## Semantic Modeling Theories

A fundamental step in the semantic treatment of programming languages in the late 1960s was the elaboration of semantic theories that helped us to understand programming paradigms or, in the case of modeling languages, modeling paradigms—not at a syntactic level but at a semantic level.

One example is assertion logic for procedural programs with its concept of an invariant. Based on an assertion calculus according to contributions from Hoare and Floyd, invariants are used to define verification rules for loops. The concept of an invariant is part of a semantic theory that can be studied independently of a particular syntactic construct just by studying the idea of predicate transformers for state transition systems according to Dijkstra. These examples demonstrate that the concept of an invariant is closely related to fixpoint definitions for iteration and show that providing semantics for a programming or modeling language in a particular semantic style also triggers the use of particular semantic theories and paradigms and thus also tools.

However, we may view this idea from a different angle: given sufficient understanding of semantic theories with potential to provide modeling constructs, the design of a modeling language should start not just from syntactic constructs but from appropriate semantic theories. The second step is to find syntactic representations for the semantic theories needed as shown in Figure 1. This is the future of language design, which is different to what we see today.

## Conclusion

As argued, the design of modeling languages should start with the design of the semantics followed by the appropriate syntax. The following key questions should be considered first:

- How should modeling and modeling languages be used and applied in software and systems engineering?
- How well and to what level of detail must the modeling language address and support the representation of specific relevant aspects and concepts?
- How is this realized, represented, and implemented by the syntactic constructs?
- Does the semantics address these constructs and concepts in an appropriate way?
- How do all these aspects support the idea of using such a modeling approach according to the modeling use cases addressed?
- What is the appropriate tooling needed to support the use cases, and what expressivity restrictions then emerge?

Such questions must be understood as requirements engineering for language design and must therefore be answered before starting a detailed design of syntax and semantics.

In the end, it is more appropriate to speak not just about modeling languages for software and system development but also about the use cases of such languages and thus classify a language more precisely as a model-based specification, design, verification, test, simulation, prototyping, programming, system implementation, configuration, or even documentation language. If, in the future, we can design concise and integrated languages, it may also be possible for one language to serve several use cases and, vice versa, for several sub-languages to contribute to the same purpose.

It will be interesting to find out how far one language, such as SysML, can and cover all these use cases. In other words, how many distinct languages or at least language variants based on a common core language are needed for effective and reliable systems development? In fact, it would be helpful for a standard, such as SysML, to state clearly what the purpose of the modeling language is and what is out of its scope. This can also be made ex-

plicit for individual sub-languages of the SysML standard.

**On the definition of languages.** Defining a modeling language requires a good knowledge of the syntactic and semantic options and possibilities. This gives us an idea of a semantic theory adequate for the real-world phenomena that we want to describe. Only then can we think about good syntactic ways to represent the respective phenomena. As described in Figure 4, we thus strongly suggest: semantics first, syntax second.

However, this does not mean that syntax is not important. It is very important to choose a syntax that is easy to deal with, comprehend, and that represents semantic properties as directly and explicitly as possible.

In some sense, this is a kind of a metaverse of modeling with a close relationship between syntax and semantics, a theory of their interaction, and ways to relate both to typical modeling tasks and use cases.

A modeling language must be designed in such a way that it does not fall into a trap that is well known in some programming languages. There are programming languages, such as PL1, which proved to be monsters, trying to comprise everything and to fulfill all potential requirements. Instead, the result was a language that was way too complicated and heavyweight for any purpose. This trap should be avoided for modeling languages. It may be better to have several slim modeling languages that are semantically coherent and match the use cases identified.

**Semantics as a basis for modeling languages.** There are many ways to describe the semantics of programming constructs. However, when designing a modeling or programming language, it might not be as important to start with how the syntax should look or which semantic techniques should be used to describe the meaning of those constructs. The question is, what are the intended use cases for the language? This is closely related to the choice of the semantic basis. Should it be possible to do proofs about the model, simulations, to generate code, and so on?

**Syntactic and semantic consistency and coherence.** For modeling languages, and specifically for a set of different modeling concepts as they are also offered by modeling languages such as UML or SysML, it is most important to be semantically consistent and coherent. Semantic coherence means that the different language and modeling constructs can be understood and used together, and consistency means that they fit together without contradictions. Checking consistency and coherence between models requires a clear understanding of how the different modeling concepts fit together—in particular, on the semantic level, such that semantically correct, comprehensive models can be worked out. Only coherent and consistent models can be composed.

**Semantics first.** Many modeling languages are designed by initially thinking about syntax, but should it not be exactly the other way around? First, concepts and modeling theories should be considered. Such concepts may be reflected by several different semantic descriptions and represented by a rich number of syntactic constructs. However, thinking about the purpose of modeling languages, the most relevant question is, what are the modeling concepts to be integrated and the necessary analysis techniques to be assisted by tools? Only then should we start to think about the syntax.

As pointed out in Figure 4, for example, it seems to be crucial to start with modeling theories based on semantic concepts before moving on to choose the syntax. This is a lesson learned from UML (see Broy and Cengarle[2]). For UML, a number of interesting concepts, including object orientation, Statecharts, message sequence charts, Petri nets, process descriptions, are mixed together on a syntactic level. Describing the semantics of the result is then very complicated. The reason is obvious: When many of the semantic concepts and theories behind the syntactic ideas do not fit together, the result is not a coherent modeling approach. Semantic coherence will hardly be achieved and defining semantics becomes an impossible challenge. Furthermore, the use cases for such languages are not met.

## Acknowledgments

We thank Conrad Bock, Sanford Friedenthal, David Harel, Klaus Havelund, the *Communications* reviewers, and especially Bran Selic for their helpful comments on earlier versions of this article.

**References**

1. von der Beeck., M. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems: Lecture Notes in Comp. Science 863*, H. Langmaack, W.-P. de Roever, and J. Vytopil (eds), Springer (2014), 128–148.
2. Broy, M. and Cengarle, M.V. UML formal semantics: Lessons learned. *Software and Systems Modeling 10*, 4, Springer (2011), 441–446.
3. Broy, M., Pepper, P., and Wirsing, M. On the algebraic definition of programming languages. *ACM Transactions on Programming Language Systems 9*, 1 (1987), 54–99.
4. Broy, M. and Stoelen, K. *Specification and Development of Interactive Systems*, Springer (2001).
5. Butting, A. et al. Semantic differencing for message-driven component and connector architectures. In *2017 IEEE Intern. Conf. on Software Architecture*, 145–154.
6. Cengarle, M.V., Grönniger, H., and Rumpe, B. Variability within modeling language definitions. In *Proceedings of MODELS 2009: Model Driven Engineering Languages and Systems*, Springer, 670–684.
7. Evans, A. et al. Meta-modeling semantics of UML. In *Behavioral Specifications of Businesses and Systems*, Kluver Academic Publishers (1999).
8. Fowler, M. *Refactoring: Improving the Design of Existing Code*, Addison–Wesley (1999).
9. Frank, U. Domain-specific modeling languages: Requirements analysis and design guidelines. *Domain Engineering*, I. Reinhartz-Berger, A. Sturm, T. Clark, and S. Cohen (eds), May 2013, 133–157.
10. Friedenthal, S. and Seidewitz, E. A look ahead at SysML v2. SysML v2 Submission Team (SST), Wasatch INCOSE Chapter (June 2020).
11. Guttag, J. Abstract data types and the development of data structures. *Communications of the ACM 20*, 6 (1977), 396–404.
12. Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (1987).
13. Harel, D. and Rumpe, B. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer 37*, 10 (2004), 64–72.
14. Hoare, C.A.R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (October 1969), 576–583.
15. Karsai, G. et al. Design guidelines for domain specific languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*. Helsinki School of Economics (2009).
16. J.-P. Katoen. The probabilistic model checking landscape. In *Proceedings of the 31st Annual ACM/IEEE Symp. on Logic in Computer Science* (2016).
17. Maoz, S., Ringert, J.O., and Rumpe, B. Summarizing semantic model differences. In *Models and Evolution* (2011).
18. Mayr, H. and Thalheim, B. The triptych of conceptual modeling. A framework for a better understanding of conceptual modeling. *Software and Systems Modeling 20*, Springer (2021), 7–24.
19. Philipps, J. and Rumpe, B. Refactoring of programs and specifications. *Practical Foundations of Business and System Specifications*. H. Kilov and K. Baclawski (eds.), Kluwer Academic Publishers (2003).
20. Plotkin, G. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming 60-61* (2004), 3–15.
21. Scott, D. Outline of a mathematical theory of computation. Programming Research Group, Oxford University (1970).
22. Selic, B., Gullekson, G., and Ward, P. *Real-Time Object-Oriented Modeling*, John Wiley & Sons (1994).
23. Weilkiens, T. *SYSMOD—The Systems Modeling Toolbox: Pragmatic MBSE with SysML* (3rd ed.) 2020.
24. Whittle, J., Hutchinson, J., and Rouncefield, M. The state of practice in model-driven engineering. *IEEE Software 31*, 3 (2014), 79-85.
25. Wolny, S. et al. Thirteen years of SysML: A systematic mapping study. *Software and Systems Modeling 19*, Springer (2020), 111–169.

**Manfred Broy** is a retired professor in the Informatics department at the Technical University of Munich, Germany.

**Bernhard Rumpe** (rumpe@se-rwth.de) is a professor in the Software Engineering department at RWTH Aachen University, Germany.