Chapter 2
Foundations

Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, Hans Vangheluwe, Francisco Durán, and Steffen Zschaler

Abstract This chapter gives an introduction to the key concepts and terminology relevant for model-based analysis tools and their composition. In the first half of the chapter, we introduce concepts relevant for modelling and composition of models and modelling languages. The second half of the chapter then focuses on concepts relevant to analysis and analysis composition. This chapter, thus, lays the foundations for the remainder of the book, ensuring that readers can go through the book as a coherent piece.

C. Talcott (✉)
SRI International, Menlo Park, CA, USA
e-mail: clt@csl.sri.com

S. Ananieva
FZI Research Center for Information Technology, Karlsruhe, Germany
e-mail: ananieva@fzi.de

K. Bae
POSTECH, Gyeongbuk, South Korea
e-mail: kmbae@postech.ac.kr

B. Combemale
IRISA – University of Rennes, Rennes, France
e-mail: benoit.combemale@irisa.fr

R. Heinrich · R. Reussner
Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: robert.heinrich@kit.edu; ralf.reussner@kit.edu

M. Hills
East Carolina University, Greenville, NC, USA
e-mail: hillsma@ecu.edu

N. Khakpour
Linnaeus University, Växjö, Sweden
e-mail: narges.khakpour@lnu.se

B. Rumpe
RWTH Aachen, Aachen, Germany
e-mail: rumpe@se-rwth.de

R. Heinrich et al. (eds.), *Composing Model-Based Analysis Tools*,
https://doi.org/10.1007/978-3-030-81915-6_2

2.1 Models, Modelling Languages, and Their Composition

In this section, we give an overview of core concepts that must be considered when composing semantics, languages, and models, and discuss how these core concepts are interrelated.

Scientists as well as engineers (including software engineers) use models to address complexity. Given this, it is worthwhile to precisely clarify what a model is. A commonly agreed-upon general definition, given by Stachowiak [Sta73], states that a *model* has three main characteristics:

- There is (or will be) an original.
- The model is an abstraction of the original.
- The model fulfils a purpose with respect to the original.

A model can be called *valid* if it fits for its purpose with respect to the original within certain *validity boundaries*. Interestingly, engineers and scientists differ in their viewpoint here [Com+20]: A scientist regards the model as invalid (or bad), if it does not describe the real world. An engineer regards the produced artefact as bad if it does not fit to the model.

We may have explicit representations of models, which can be defined using natural language or a more formal modelling language. Existing modelling languages can be classified as general-purpose modelling languages, such as the *unified modeling language* (UML) [BRJ98], and *domain-specific modelling languages* (DSML) [Kle08]. For example, software developers regularly use class diagrams to define data structures, concepts of the real world and their relations, and also technical architectures within the software.

The advantage of such *explicit* models is that they can be used as documentation, be subjected to different forms of analysis, or even be used as a source to produce some output, including code generation or 3-D printing. The specific focus that a modelling language usually has may be a burden for the modeller, because of the restrictions that it imposes, but also enables many smart constructive and analytic algorithms based on that language. For example, state machines can be checked for completeness and determinism, the *structured query language* (SQL) provides efficient database retrieval and storage based on E/R-models, etc.

P. Scandurra
University of Bergamo, Bergamo, Italy
e-mail: patrizia.scandurra@unibg.it

H. Vangheluwe
University of Antwerp, Antwerp, Belgium
e-mail: hans.vangheluwe@uantwerp.be

F. Durán
University of Málaga, Málaga, Spain
e-mail: duran@lcc.uma.es

S. Zschaler
King's College London, London, UK
e-mail: szschaler@acm.org

It is also possible that models, instead of being expressed in a certain modelling language, are encoded directly within a general-purpose programming language, like C++, Python, or Java. These models are typically used, for example, in simulations, such as of phenomena in climate and weather, at the atomic level, in cell biology, or in the wider universe. In this case, the sole and only form of analysis is through direct execution and an examination of the resultant execution trace. The availability of code also allows the possibility of checking certain coding properties, as type consistency or the correct handling of exceptions. Tools like Coverity [Syn] or CodeSonar [Gra] provide quite sophisticated forms of what is typically called (static) program analyses. The following discussions concentrate on modelling languages and their use for the definition of models. A discussion of the use of modelling languages in the construction of simulation models can be found in [ZP20].

2.1.1 Types of Models and Their Role in Analysis

Various types of models [Lee18] and the roles they can play [Küh16] are described in the literature. Here, we adopt the distinction made by Combemale et al. [Com+20], who consider three types of models: engineering, scientific, and machine learning models.

An *engineering model* is used to specify and represent a targeted system [Lee18]. It drives the development of the system to be built by specifying concerns such as, e.g., braking and obstacle avoidance in on-board control systems for autonomous vehicles, traffic management models, information systems, or business rules. Engineering models are typically used as a means to develop a physical system, a software-based system (including behaviour, structure, and the interaction of the system with its context), or both (e.g., cyber-physical systems). Engineering models can be described using both domain-specific and general-purpose languages.

A *scientific model* is a representation of some aspects of a phenomenon of the world [GL16]. Scientific models are applied to describe, explain, and analyse the phenomenon based on established scientific knowledge defining a theory. A theory provides a framework with which models of specific phenomena and systems can be constructed. Scientific models are used in various application areas ranging from climate change models, to electromagnetic models, protein synthesis models, or metabolic network models. Typical examples include continuous, equation-based formalisms like differential equations, or discrete-event models.

A *machine learning model* is created by automated learning algorithms based on sample data (i.e., training data) to make predictions or decisions without being explicitly programmed for the task at hand. It approximates the conceptual relationship between a given input and the expected, or a priori unknown, target output. Machine Learning models can be applied in various application areas such as image classification, feature extraction, defect density prediction, language translation, or motion planning of robots. Common formalisms include neural networks, Bayesian classifiers, and statistical models.

According to [Com+20], a model can play several roles with respect to its purpose: It can be descriptive, prescriptive, or predictive.

- A model plays a *descriptive role* if it describes some current or past properties of the system under study, facilitating understanding and enabling analysis.
- A model plays a *prescriptive role* if it describes properties of the system to be built, driving the constructive process—including runtime evaluation in the case of self-adaptive systems.
- A model plays a *predictive role* if it is used to predict properties of the system that one cannot or does not want to measure.

Each type of model can play several roles, which determine whether and how the model is used in analysis. A scientific model is descriptive first and then may become predictive, e.g., to support what-if analyses [Bru+15]. The model may also become prescriptive, e.g., if embedded into a socio-technical system. For example, a prescriptive model of a decision-making tool for climate change using a predictive simulator based on a descriptive scientific model of the earth's water cycle [Com+20].

An engineering model typically starts by being descriptive and then, at design time, is refined and transformed into a prescriptive model. Then, once the system is built as prescribed, the model becomes descriptive again as a kind of documentation [Hei+17]. An engineering model may also be used as a predictive model. For example, a system architecture model can be applied to predict the performance of a specified system configuration [Reu+16].

A machine learning model is typically used in a predictive role to infer new knowledge based on some hypothetical input data. It might also be descriptive of a current or past relationship, or prescriptive if machine learning results are used for decision-making [Com+20]. For example, we may have a prescriptive model of a smart factory where a predictive model is used to make decisions about production plans based on descriptive historical data.

2.1.2 What Is a Modelling Language?

According to [Com+16], a *modelling language* defines a set of models that can be used for modelling purposes. Its definition consists of

- Syntax, describing how its models appear,
- Semantics, describing what each of its models means, and
- Pragmatics, describing how to use its models according to their purpose.

This is a rather general definition which can be realised in various ways. Graphical, tabular, and textual forms of syntax are possible. The semantics can, for example, be defined in denotational form [Sto77], where a semantic domain is mathematically defined and a mathematical function relates syntactic elements to elements of the semantic domain. The semantics could also be defined by explaining the execution effect of model elements, for example, using abstract machines.

Models need a precisely defined semantics. Semantics describes the precise *meaning* [HR04] of each well-formed model in terms of the *semantic domain*, which is generally well understood. We also speak of a *formalism* or *formal language* instead of a *modelling language*. Such a semantics is the formal foundation for understanding whether a model is correct; that is, whether desired properties are satisfied. For example, the set of words over an alphabet can be used as a semantic domain for state machines. A concrete state machine can then be mapped to the subset of accepted words.

There are different formalisms and formal methods for the specification of well-defined semantics, such as rewriting logic semantics [MR04], small-step/structural-operational semantics [Plo04], or big-step/natural semantics [Kah87] or their extensions to distributed, event driven systems [BS01]. In many cases, however, no explicit semantics is available, or we may say that there are different forms of ad hoc semantics. For instance, despite different attempts to provide a formal semantics, the semantics of Python continues to be defined primarily by the behaviour of the Python interpreter, just as the semantics of Java is provided by its virtual machine.

The main purpose of software is “to do”. Therefore, many modelling languages concentrate on software and the behaviour it specifies. However, it is worthwhile to mention that semantics should not be confused with behaviour as such, because purely structural languages, such as class diagrams, also have semantics, which in this case is the set of possible object structures.

It is also worthwhile to note that the semantics should not be confused with a *real world interpretation*. For a precise study of the phenomena of a formal language, the semantic domain should be a mathematically well understood, precisely defined construction. It is then up to us to *interpret* this in the real world. For example, words over an alphabet may be interpreted as sequences of human actions in workflows, sequences of messages over a communication channel, or sequences of produced physical component parts in a production line.

A *sound* semantic definition is very helpful to understand what shall be analysed and what the desired outcomes of analysis techniques are. This is true for binary results of analyses, but also for quantifiable results that rely, for example, on statistical considerations. In practice, an explicitly defined semantics is not always needed. Sometimes the language designers already have a good informal understanding and can directly encode the desired properties into algorithmically executable analysis techniques. However, if the domain is complicated, not very well understood, or if the DSML is newly defined, then an intermediate step consisting of mapping the syntax of the language into a semantic domain, before designing executable analysis techniques, has proven very helpful in practice. Such a mapping into a semantic domain can indeed explain the desired and technically implemented results of an analysis technique much better. This even holds if it is not formally defined, but used as a shared understanding for a formal language. The following examples illustrate the use of semantic domains (traces, object-diagram structures, or Petri nets) for different analysis problems.

- The failure rate of state machines can be well explained using a set of traces. In this case the syntax is the state machines, and the semantics its traces. The analysis technique needed is the efficient examination of a representative finite set of traces or a BDD-like[1] integrated representation of all traces.
- The coverage of test sets can be well explained over an appropriate minimal set of object structures that can be derived from a class diagram. In this case the syntax is the class diagrams, and the semantics its object structures. The analysis technique needed is a monitoring of the tests and a finite grouping of relevant object structures into equivalence classes.
- The set of reachable states of a machine can be understood using a mapping from state machines into Petri nets. In this case the syntax is the state machines, and the semantics its Petri net representation. The wide range of tools available for Petri nets provide the needed analysis technique.

According to the above definition of models, a model has a purpose with respect to its original. In practice, a model can be used for more than one purpose, typically associated with the above-mentioned model roles, and such purposes may change during the model's use in various activities of a development process. For example:

(a) A business analyst uses a model to capture and convey requirements and other related information known about the system to be developed.
(b) A developer uses the same model to constructively implement a system fitting to the model.
(c) Quality assurance may use the same model to analyse different quality properties.
(d) Testers use that model both to identify potential problems and derive tests.
(e) Engineering models might be used to realise a digital twin that collects data, provides services and, to some extent, also controls the physical twin [Bib+20].

While for all those the semantics of a model is the same, they use the model in rather different ways and therefore also need rather different functionality centred around the model.

2.1.3 Metamodels

Tools need a suitable representation of models to manage them. An approach that has proved itself useful for constructive as well as analytical tools is metamodelling technology [Gro06].

[1] BDD stands for Binary Decision Diagram.

It is a core idea of *model-driven engineering* (MDE) to also use models for explicitly defining modelling languages: this is called *metamodelling*. Let us borrow the following definitions of metamodel and conformance from [Com+16].

A *metamodel* is a model describing the abstract syntax of a language. It is commonly agreed that a metamodel is usually defined as a class diagram, very similar to UML class diagrams. A metamodel therefore describes a set of object structures, where each of these object structures describes the abstract syntax and therefore the essence of a model, which is needed for analysis, code synthesis, or other development activities.

A model *conforms* to a given metamodel if each model element is an instance of a metamodel element. Such a model is considered valid with respect to the language represented by the metamodel. It is a big advantage of the metamodelling approach that exactly the same metamodel can be used within a tool both to represent the abstract syntax and to operate on such model, for example, for code generation or for the application of some analysis technique (see Chap. 11 of this book [Hei+21] for an example). A metamodel therefore serves a dual purpose, defining the language and all the models in that language (i.e., the syntax), and serving as basic infrastructure for tooling. The class diagram however only captures the abstract syntax, which needs to be augmented by a concrete representation, usually in diagrammatic, tabular, or textual form.

2.1.4 Property Models

In model-based analysis, we are interested in understanding certain properties of the system under study. When the property of interest can vary, such properties must be made explicit by the developers and then provided to the tooling. Given this, it is necessary to give the desired properties a precise semantics in addition to the models. For that purpose, we distinguish between *properties* that talk about the semantic domain of the model, e.g., the set of possible system runs, reachability of states in a model, or climate behaviour, versus pure *syntactical properties*, such as readability or cyclomatic complexity of code.

As a consequence of these considerations, a semantic property is itself a kind of model, which we call a *property model*. The property is then the set of all elements of the semantic domain that satisfy the property. Formulated in set-theoretical form: A model fulfils a property, exactly if the semantics of the model is contained in the semantics of the property. Or, formulated in logical form: The semantics of the model implies the semantics of the property.

2.1.5 Two Dimensions of Model Compositionality

Models need to be compositional along two dimensions, namely concerns and subsystems. To tackle the complexity of systems development, it is often necessary to use diverse models describing different aspects or viewpoints of the system as a whole or of subsystems.

The separation of concerns is important for enabling contributions by different subject-matter experts, and enables parallel development and evolution. This is why modern modelling languages, such as UML [BRJ98] and SysML [Gro12], provide a number of different sublanguages, allowing the modeller to concentrate only on certain aspects of the system.

Model reuse is only possible if the models are developed in independent, relatively encapsulated pieces. Building models as encapsulated pieces implies a mechanism and opportunity for composition, and building up a description of the overall system. For example, we may use state charts to describe the behaviour of individual components, and then combine them into a model of the system as a whole. These behavioural descriptions of components can then be reused to build models of different systems.

In summary, the two dimensions of model composition are:

(a) Within the modelling language, models are semantically composed to produce larger specifications. This form of composition usually goes along with the composition of the system components.
(b) Models of different aspects of the same component are composed to give a more complete description of this component.

2.1.6 Models of Context

In many cases, in addition to the models of the system or component to be built, we require a model of the *context* of the system or component.

In systems modelling, the term “context” refers to models which are needed for system construction and analysis but which do not describe entities to be built. Instead they describe the environment of the system. Considering contexts is important as they affect the system to be built for two reasons:

(a) If the implementation of the system can rely on certain *assumptions* about how it is being used, then certain internal optimisations become possible.
(b) When using the implemented subsystem/component, the context model actually gives restrictions on how it can be composed into a larger system.

If we lift this understanding of context to the metamodel (the language definition level, Sect. 2.1.3), we also have to draw a line between the artefacts to be manipulated and the additional *context* information. As an example, given all potential executions of a system, a context model may define the subset of executions that need to be analysed. In addition, one could also consider platform-specific parts of the semantics definition (like scheduling policies) as a context model which act as a parameter to the analysis. Alternatively, such information could have also been considered as part of the model's semantics. The choice depends on the specific questions to be studied and on the available analysis tools.

In the case of security analysis, for example, the context model may include an attack model (e.g., the Dolev–Yao model [DY83]), giving the capabilities of attackers of concern. When analysing robustness or resilience against faulty or inaccurate sensors or actuators, a context model might include fault models. In the case of a multi-agent system, analysis might focus on a single agent to reduce analysis complexity, using a context model that includes a model of the remaining agents, suitably abstracted.

2.1.7 Model and Language Composition

Composition of models and the languages used to describe them is key to coping with the complexity inherent in modelling diverse aspects of large systems, possibly modelled in models expressed in heterogeneous languages. This, therefore, implies that also languages need to be composed.

In general, when composing languages, we need to understand what to compose, given as shown in Sect. 2.1.2, that a language is defined by syntax, semantics, and pragmatics. Assume you have one syntax definition and two different semantic definitions. For the purpose of composition, we might view these as two different languages. However, having one syntax and semantics definition, and two pragmatics descriptions, we would still consider this to be one language, with two ways of using it. Consequently, when considering composition, we would treat syntax and semantics definitions as constitutional parts of the language (any change of either will change the language), while the pragmatics description would not be part of the language and, hence, not subject of language composition.

Looking at the efforts to standardise UML in recent years, we can observe that a clear, precise, and well integrated semantics for such a language is not easy to achieve. This is partially due to the complexity of UML itself, partially due to political problems, because different driving forces have different understandings and interpretations, but also partly due to the overwhelming desire that UML should cover every domain of software systems. Actual realisations of components may differ in multiple details, for example in communication forms, timing, interaction of threads, sharing of memory, etc. If UML is mainly used for communication between developers in the form of "paper-based" models, this lack of semantics is not necessarily harmful. For sophisticated analysis techniques, however, this is a strong impediment.

For an advanced and potentially integrated form of composite semantics, it is necessary, as a consequence of the above discussion, to also think of composition of models of different aspects and, therefore, as well on the modelling languages used for describing these aspects. Instead of a one-size-fits-all language, such as UML, a feasible alternative could be to use small individual modelling languages, allowing us to describe small and focused models, and then integrate the models by integrating the analysis techniques or the results of the analysis algorithms as described in Sect. 4.4.2 of this book [Hei+21].

Knowing that to manage complexity it is important to decompose the problem, so different aspects can be addressed individually, as well as to decompose system models into models of subsystems, it is clear that we also need mechanisms to compose models, including those written in different heterogeneous languages.

Forms of Model Composition Although there is work on model composition, coming up with a complete classification of forms of model composition is challenging, because the form of composition very often depends on the form of the models and the aspects they describe. For example:

- Class diagrams can be merged (see, e.g., [Obj17, DDZ08]).
- State machines can be composed using the cross product for synchronous communication, but there are also other forms of composition when allowing asynchronous communication or feedback (see, e.g., [LV03, Chapter 4]).

For software code written in typical programming languages (e.g., Java or C), composition of different code components is typically not performed at the source-code level. Instead, composition is normally delayed to the binding stage. For example, in virtual machines this only happens when loading the compiled code. Composition is conceptually clearly understood on the source-code level, but modularity allows to defer the actual composition to a very late stage, which supports agility of development. In contrast, state machines have composition techniques that are applied directly on the state machine models. This is true for many other modelling languages, too.

It is important that analysis techniques are designed to be as modular as possible. They can then be applied to certain model subsets, for example, the models of a subsystem, independently of other subsystems. This will lead to more efficient analysis execution, delivering results more quickly, potentially even immediate feedback during editing, and to a better reuse of the analysis results when composing the overall system (see also Chap. 7 of this book [Hei+21]). However, some analyses cannot be modularised. These analyses need a holistic view of the overall system model and an understanding of how the system will be used in its context, which therefore also needs to be explicitly modelled.

Forms of Language Composition Composition of models within a given language is already a challenge, but has been addressed at least in formal methods and also by a variety of tools. Composition of models expressed in different languages is equally important, in particular in the context of holistic analysis techniques.

When composing models expressed using different languages, some consistency checks are necessary. One needs to ensure that symbols defined in one language, for example, classes in class diagrams, are consistently used in the other languages, for example, in object diagrams or performance models. These checks can in many cases be executed automatically.

Composition of models may be interpreted as the integration of models into one single uniform model, but it may also mean that models of subsystems or concerns are somehow coordinated. For example, state machines that describe the behaviour of individual components of a component diagram may be connected through component diagram channels [SGW94] in various ways, and their composition is of course dependent on the component diagram structure.

An alternative to coordinating models in different languages is to compose the languages and then compose models within the resulting language. Given that metamodels are class diagrams, we have several alternatives for language composition:

- We can use merging algorithms for class diagrams to get an integrated metamodel. This, however, involves a lot of design choices, especially when the language concepts that should be the same are technically realised in different forms in the different metamodels. See, e.g., [Cla11], and works on Concepts, Templates and Mixin Layers by de Lara and Guerra [LG10], Melange [WTZ10], model amalgamation [Dur14, Dur+17] or the GTS Morpher in Chap. 9 of this book [Hei+21].
- We can define mappings between metamodels that would allow to translate the complete model of one language or at least certain parts of one model into a model of the other language. This approach is more decentralised, because it does not need a one-fits-all integrated metamodel. However, it leads to redundancy on the model level, which in turn leads to issues when evolving models. Changes in the different models need to be synchronised. See, e.g., [Hu+11, GS18, Hid+16] for an overview on bidirectional model transformations.
- We can define consistency relations between the metamodels. These relations can, however, only be used to check consistency, but are not helpful in constructive adaptations.

These forms of composition may also be mixed in different ways. And of course, we may have cases in which neither the metamodels nor the models are integrated at all. In these cases, the execution of the models may still be synchronised both in time and events and potentially also in shared data. Chapter 9 of this book [Hei+21] describes a form of metamodel composition that includes a composition of the semantics.

2.1.8 Model Transformations and Transformation Models

In order to explicitly capture relations between models, we need to establish a *transformation model* [Béz+06], which, in turn, obeys its own *transformation language*, often specified in form of a *transformation metamodel*.

Transformation metamodels typically connect a source and a target language (metamodel) through explicit constructs for expressing relations (e.g., QVT-Relational [OMG16]) or algorithmic translations (e.g., QVT-Operational [OMG16] or ATL [Jou+08]). A transformation model specifies a set of *model transformations*, one for each acceptable combination of input models. It is, therefore, sometimes referred to as a *model transformation specification*.

Relational transformation models can potentially specify model transformations in different directions, including transformations used to bi-directionally synchronise different models [Hu+11, GS18, Hid+16]. Algorithmic transformation models typically fix a particular transformation direction.

Many transformation languages come with dedicated engines for efficiently "executing" a given transformation model; that is, for instantiating the corresponding model transformation for a given set of input models.

Model transformations have been used in many different areas, including model translation, model composition, refinement, etc. Surveys [CH06, REP12, Men13] provide classifications of model transformation approaches and languages, showing the features of the most prominent ones. More recently, in [Kah+19], Kahani et al. provide a detailed overview of the state of the art in model transformation techniques and tools by presenting a catalogue of 60 metamodel-based transformation tools, which are categorised in accordance with several attributes.

The correctness of model transformations is key for MDE. Their correctness is even more important if model transformations are used to compose models and interoperate analysis tools, since the validity of such analysis rely not only on the analyses or analysis tools themselves but also on the translations to which models and analysis results are subjected. [CS12] and [Amr+15] present exhaustive reviews of the literature on the verification of model transformations analysing the types of transformations, on the properties that the different existing techniques verify, and the verification techniques that have been applied to validate such properties. [RW15] also surveys research on model transformation verification by classifying existing approaches based on the techniques used (testing, theorem proving, model checking, etc.), level of formality, transformation language used, and properties verified.

Related to the verification of model transformations, we have testing and static analysis of such transformations. For example, surveys such as [Mus+09, Bau+10, SCD12] present views on the state of the art in the area of model transformation testing. Sánchez Cuadrado, Guerra, and de Lara make an interesting proposal in [CGL17] for the static analysis of model transformations. They present a method for the static analysis of ATL model transformations (also discussed in Chap. 12 in this book [Hei+21]). Their goal is to discover typing and rule errors using static analysis and type inference.

Formally and systematically defining the semantics of transformation languages is important to ensure consistent and predictable execution of transformations. Different approaches have been taken for defining the semantics of model transformations. For example, in [TV10, TV11], Troya and Vallecillo give a formal semantics of the ATL 3.0 model transformation language [Jou+08] using rewriting logic and Maude. [RN08] translates model transformation definitions in the QVT (Query/View/Transformation) [OMG16] language to a graph production system, thus providing a graph transformation-based semantics to it. Guerra and de Lara propose in [GL12] a formal, algebraic semantics for QVT-Relations check-only transformations, defining a notion of satisfaction of QVT-Relations specifications by models. In [CS13], Calegari and Szasz present a formal semantics for the meta-object facility (MOF) and QVT-Relations languages based on the Theory of Institutions. With this approach, the semantics given reflects the conformance relation between models and metamodels, and the satisfaction of transformation rules between pairs of models. Indeed, the theory facilitates the definition of semantic-preserving translations between the given institutions and other logics which will be used for verification.

2.2 Analysis and Analysis Composition

The essence of analysis is to answer questions about properties of interest of a system under study. However, an analysis generally does not reason directly about such a system, but instead about a model of the system. An analysis that reasons about a model of a system under study is called a *model-based analysis* in this book. To allow a model to be used to answer questions about the system it represents, the properties of the model need to adequately reflect those of the system under study. Models can be used to represent the structure of a system (i.e., its parts and how they are connected), the behaviour of a system (i.e., what it actually does when it is executed), the interaction of the system with its context (i.e., other systems, the physical environment, or humans), quality aspects of the system (e.g., performance, code complexity, or energy consumption), or a combination of these. Similarly, properties can target structure (e.g., to reason about connectivity or potential flow of information), behaviour (e.g., to reason about the correctness of what the system does, or how it interacts with other components or its environment), or quality aspects (to determine if the system guarantees a desired quality).

From a semantically and precisely defined relation between models and properties, we need to derive practical algorithmic analysis techniques, that effectively compute the answer to a query on the composition of a system model and its context. We define an analysis as the following judgement:

$$M, C \vdash_T Q \rightsquigarrow A$$

stating that the query Q on a system model M in the context C leads to the answer A using the technique T. The *satisfaction relation*, often denoted by $M, C \models Q$ is an instance of such entailment where $A = \mathsf{true}$. From the literature, we know a number of such precisely defined satisfaction relations, such as

(a) Logical implication for various forms of logic [Tom99],
(b) Refinement and various forms of (bi)-simulation relations [Mil89],

(c) Satisfaction of temporal logic formulas by automata [CES83], and
(d) consistency checks for object structures with respect to given data structure definitions (e.g., class diagrams) [RG00].

Analysis techniques can be categorised along different dimensions, including automation degree, time, result type, purpose, quality, and composition of analysis, that we will discuss in the following.

2.2.1 Automation Degree

The first dimension for categorising analysis techniques is the degree by which they can be automated by tools. An analysis can be carried out fully automatically by a tool, semi-automatically (i.e., interactively or tool-assisted), or manually, depending on the kind and complexity of the analysis problem. The reason for its level of automation is diverse. While some analyses can be performed in a fully automatic way, others are computationally expensive or undecidable, which makes them less amenable to full automation. Software that partially or fully automates an analysis is called *analysis tool* in this book. There are multiple alternatives for analysis automation, including:

- Typically, analysis such as type checking and well-formedness checking can be carried out fully automatically.
- In the formal methods community, there are several techniques and their supporting tools such as model checking [CES83, KNP11, Ben+95], symbolic execution [Kin76, How77], automated theorem proving (e.g., Hoare logic-based verifiers [Hoa69]), and satisfiability modulo theories and solvers [Bar+09] to analyse a system automatically.
- In some cases, we may perform analysis on specific aspects of systems. For example, at the programming level, data-flow and control-flow analyses are techniques that allow the user to perform checks on the dynamic aspects of our programs, checking, for example, whether a storage location has been initialised before it is being used, or whether a program leaks information.
- Simulation is another example of automated analysis. Simulation typically works with specific scenarios and supports detailed analysis of functional and non-functional (e.g., performance) aspects of a system's behaviour.
- While the satisfaction of many properties can, in principle, be checked automatically, this is not always possible efficiently, for example, when using backtracking-based exponential algorithms. Model checking of large models is a prominent example. Large models cannot be model-checked automatically due to the state explosion problem [Cla+11]. As a result, analysis algorithms sometimes stop with an undetermined result. In some cases, a combination of various techniques might be used to perform the analysis, such as abstract interpretation, refinement checking, and modular or incremental analysis. Another alternative is to provide a partial analysis result. For example, *bounded model checking*

explores the system's state space up to a specific depth, and provides a valid result only up to that depth.

- It may be that the analysis judgement cannot be automatically established, but needs help from the developers/designers. For example, in program verification, analysis users provide explicit hints and assertions, such as loop invariants, to support the analysis. In general, verification systems based on theorem proving—for example, Isabelle [NPW02] or PVS [OS08]—fall into that category where the tool can often provide semi-automatic proof assistance, but active proving effort is needed. Another common example are verification tools driven by developer-provided annotations, written using notations such as the *Java Modeling Language* (JML) [Bur+03] or Spec# [BLS05].
- Finally, there are informal but systematic methodological techniques. In this case, explicit reviews of models according to the defined properties are carried out—for example, certification processes in which the properties are met if all the reviewing criteria are determined to hold. Of course, certifiers may be assisted by all kinds of analysis tools to do this.

2.2.2 Design Time vs. Runtime Analyses

The time at which the analysis takes place is another dimension to categorise analysis techniques. In this dimension we consider *design-time*, *runtime*, or *hybrid* analysis techniques.

Design-time analysis techniques deal with analysing models of the system at design and compile time. Such analyses can be done either at the program level or using an abstract model of the system for analysis. Design-time analysis is helpful to identify violations before they occur, and can help to find design problems early, when they are easier or less expensive to fix. However, in some cases, analyses must be postponed to runtime. This is due to (i) the nature of the analysis, e.g., in case of undecidable satisfaction relations or the state explosion problem in model checking, or (ii) the lack of detailed information at design time, as a design-time model often describes the system at an abstract level. Although analysis can be done ad hoc, design-time analysis is often done using more formal, semantics-based techniques. This includes the semantics techniques already mentioned above, as well as others, such as abstract interpretation [CC77] or matching logic [RES10]. Model-based software development is an example where various model transformation and analysis techniques are employed to design and analyse a software system [Voe+13, Kus+17].

A *runtime analysis* is based on a notion of *execution* of the model or system. To this end, execution traces or similar elements are included in the semantic domain of the modelling formalism, which may be augmented with information needed for the analysis (e.g., security properties in a security analysis). At runtime, the desired properties can be checked against individual system executions, against an abstract model of the system, which is maintained and updated at runtime to reflect the changing behaviour of an adaptive system (called models@runtime) [BGS19], or against the events that occurred during execution. Runtime verification [LS09] is an example of runtime analysis where an execution of the system is analysed—for example, to check simple assertions, temporal logic formulas, automaton-based properties, etc. Testing is another example of dynamic analysis [Bin00].

In a *hybrid* approach, the results of design-time analysis are used to generate a monitor that runs along with the system, observes its behaviour, analyses it, and possibly dynamically adapts itself (see, e.g., [GS02, Ald+19, KS18]).

2.2.3 Quantitative vs. Qualitative Analyses

As a third dimension, we may categorise analysis techniques according to whether the answer to the query is binary (true or false), or quantifiable. Queries about whether a given predicate holds of a system under study are the main example of qualitative analysis. The predicate can concern a specific scenario, or it can concern "all" executions in some class of contexts. A simple example is analysis of the function computed, such as the *isSorted* property of the list returned. Another example is an analysis to check that a system rejects illegal input (that is formally specified). Checking that specified state invariants hold is another example.

While many types of analyses can be formulated as satisfaction relations that are binary, in practice there are also a lot of queries that deal with quantifiable properties. For example, the quality of service needs to be measured by the up-time of the system, by behaviour under load, or by meantime of delay for transport of data, video and speech. Properties are then often defined using probabilities, intervals, or numbers (representing measurements).

Quantifiable properties usually lead to analysis techniques that also produce quantified results, that can be thought of as a degree of satisfaction. This opens the possibility for different system models to be compared by *how well* they satisfy a certain property, enabling an optimisation-based approach to software design, such as is explored in the field of search-based software engineering [HMZ12] or, in the modelling context, search-based model-driven engineering [BSA17, Joh+19].

Often, the quantities are expressed in the property, using, for example, interval ranges for some quantity or bounds on probabilities, while the checking itself is binary. For example, timing can be modelled using formalisms including timed automata [AD94], or timed transition rules [ÖM07], with properties expressed with timed temporal logics that are checked by model checkers such as UPPAAL [BDL04] or Maude [Cla+07]. To model properties about performance, or check the probability of events occurring or conditions holding, probabilistic models can be used. Properties of such models can be expressed in probabilistic variants of temporal logics and checked by stochastic model checkers such as Prism [KNP11]. Precision of analysis can be traded for scalability by using statistical model checking using tools such as PVeStA [AM11]. Statistical methods rely on sampling the execution space, for example, using simulators. Simulation is a technique for quantifying satisfaction executing the system model on exemplary input data and simulated interaction with the system's context. The analysis collects aggregated data about the overall system behaviour in the form of traces, which are then used to quantify satisfaction.

2.2.4 Purpose of Analysis

A further dimension for categorising analysis techniques is their *purpose*. Seeing "analysis" as answering queries about systems under study, we identify the following possibilities:

Structural analysis is concerned with analysing the system at the structural level. A *structural model* describes the elements of the system and their relationships, e.g., a call graph describes the methods of a program and their invocation relationships, or a component model (e.g., BIP [BBS06]) allows us to express the architecture of a system as a set of modules and their interactions. From such models, dependency relations can be derived. Coordination models often work at the level of components, organising the interactions. The underlying graph structure may give useful insights, for example, identifying hubs or components that mediate interactions. Graph rewriting and transformation systems [Roz97] is a class of techniques often used to analyse the systems structure.

Behavioural analysis is concerned with properties of system executions. A *behavioural model* allows analysing a system's runtime behaviour—its interactions and results—by reasoning about properties of the model's semantics. For example, "does the light go on when the door opens?"; "does the data store correctly save and retrieve data?"; or "does a warehouse robot pick the correct packages?" The analyst may only be interested in behaviour in a specific set of conditions, such as a particular set of data, or a particular region of the warehouse. These restrictions can be expressed as a context model to compose with the system model, or simply specified in a configuration file. The first question above might be analysed, e.g., using a dynamic dependency analysis of sensor and actuator events. The second question might be treated as a verification problem: The analyst would develop a formal model of the data store, express the properties as formulas, and use a model checker or theorem prover to show that the model satisfies the corresponding formulas. For answering the third question, the analyst might choose testing, providing a variety of tasks to test the robots capability.

Quality analysis is concerned with assurance of quality properties of a system (often also called non- or extra-functional properties, as for example defined in the ISO/IEC 25010 [25011] standard). Quality properties include performance, reliability, or availability. Similar to testing, the analysis of such quality properties depends on the expected usage of the system. For example, in the analysis of security aspects, attack models describe the "usage" of the system by an attacker. Safety analysis requires a definition of states deemed unsafe and conditions that could lead to safety violations if the system is not properly designed and implemented. Effects of a successful security attack could lead to unsafe conditions, unavailability, or other quality failures. Thus, combining analysis of, e.g., security with other quality properties is important.

In addition, information about the execution environment is needed to interpret executions regarding the quality property under analysis. Examples are the speed of hardware resources in case of performance, or the security guarantees provided in case of security analysis. In principle, such models of the execution environment could be seen a part of specific semantic models for quality analyses. However, as they describe environmental factors, not being part of the system being built, and as their change does not change the system (but the analysis results), we consider these environment models also as context [Zsc09].

Structural/behavioural co-analysis The system's structure and behaviour can affect each other, e.g., if component interaction is constrained by the system structure, this will affect component's internal executions and, consequently, the whole system's execution. While structural analyses may operate only on syntactical elements and behavioural analyses concern system executions, co-analyses of structure and behaviour consider both aspects. HPobSAM [KKS19] is a model to co-specify the system's behaviour and structure using graph transition systems. Chapter 9 of this book [Hei+21] presents a tool for the composition of DSMLs defined with both structure and behaviour (defined with graph transformation rules). Other formalisms, e.g., graph grammars [Roz97] and rewriting logic [Mes92, Mes12], can also nicely express both structure and behaviour.

Table 2.1 summarises, for each kind of analysis, which kind of models are required. Table 2.2 shows, for different analysis types, the information to be provided by context models.

2.2.5 *Correction and Counterexamples*

Knowing that a satisfaction relation has not been met, or that the degree of satisfaction is not high enough, is only half of the solution. We also need to understand how to improve the model and the implemented system in such a way that the desired properties are met. Again, we can see different categories of assistance here:

Table 2.1 Kinds of analyses and their required model kinds

	Syntax	Semantics	Context
Structural	x		
Behavioural	x	x	(x)
Quality	x	x	x

Table 2.2 Examples of different analyses and their required context models

Analysis	Kind	Required information in context models
Simple dependency analysis (e.g., static component dependencies)	structural	–
Advanced dependency analysis (e.g., slicing, points-to)	Behavioural	start item for analysis
Verification	Behavioural	Fixation of parameters in semantic specification, e.g., platform-specific scheduling policies
Testing	Behavioural	execution environment, test case specification
Performance, reliability	Quality	Usage profile, deployment and resource descriptions, description of external service quality
Safety	Quality	Definition of set of safe/unsafe states
Security	Quality	Attacker model, model of platform security
Maintainability	Quality	Change propagation rules, seed modification

- The analysis technique tells us that the satisfaction relation is not met, but gives no hint beyond that.
- If the satisfaction relation is not met, we at least get hints where the problem is located. This may be, for example, specific elements in the model which contributed to the problem, or the places where certain desired invariants have not been met the first time. Another example is counterexample generation by model checkers [HKB09], where when a property is not satisfied, a counterexample is usually generated and provided to the user that can help identify the reason of violation. A counterexample is usually an execution trace that violates the property.
- As a result of the analysis, we not only get the location where the problem arises, but also a list of suggestions, what can be done to correct the problem. This is typically the case in an integrated development environment (IDE) that checks context conditions already while source code is being edited, and suggests a list of possible corrections on the fly. This, however, is more effective if the problem can be relatively easily localised. There is a lot of experience on what the typical error sources are in many different cases: wrong type chosen for a variable in a program; unsatisfiable trigger condition of a transition in an automaton; or not enough redundancy on the available compute nodes in a performance model.
- The last category of analysis techniques not only identifies flaws, but also automatically corrects them. *Automatic program repair* techniques (see [GMM19] for a survey) fit in this category, where several techniques are used, e.g., a generate-validate-test approach to generate a fix that will be tested before being accepted (e.g., GenProg [Gou+12]), or a semantics-driven approach where formal approaches are used to synthesise a patch (e.g., Angelix [MYR16]).

See Chap. 7 of this book [Hei+21] for an in-depth discussion of how analysis results can be used.

2.2.6 Quality of Analyses

Two important concepts related to the quality of an analysis are *soundness* and *completeness*. Recall that analysis was earlier defined as the judgement $M, C \vdash_T Q \rightsquigarrow A$, where analysis technique T is used to answer query Q over the model M in context C. Here, we focus just on those cases where the answer A is either *true* or *false*. Thus, the analysis is answering the question whether the property Q holds of M in context C. To determine the soundness or completeness of T, a "ground truth" is needed. One form of ground truth is a semantics of both models and properties. Thus we assume a mapping $[[M]]_C$ giving the meaning of M in context C as an element of a semantic domain D, and a mapping $[|\ Q\ |]$ of properties to subsets of D.

T is *sound* if T derives *true* as the answer to Q only when Q is *actually true* of M in context C. That is, $[[M]]_C \in [|\ Q\ |]$. An analysis technique T that incorrectly derives *true* in this case—that is, that says that the answer to Q is *true* when it is actually *false*—is *unsound*. To ensure soundness, T will answer *false* in cases where T cannot prove *true*. Another option is for T to only answer *false* when Q is definitely *false* for M in C. Such an analysis is said to be *complete*. For instance, if T is an analysis technique that checks for deadlocks, M is a model of a concurrent system, and Q is the property "M is deadlock-free in context C", a sound version of T will not answer *true* when it cannot prove that M is deadlock-free in context C, meaning it may potentially answer *false* in some cases where M actually is deadlock-free. A complete version of T will not answer *false* when M is deadlock-free in context C, but may potentially answer *true* when M is not deadlock-free (i.e., when it may deadlock). For some properties, it may be difficult or even impossible to fully establish ground truth. Measuring reliability or performance are examples. How does one know if a real system satisfies 99.9% up-time, or if the system responds within 1 s with probability 0.95?

Most analysis techniques T cannot be both sound and complete, due to the undecidability of the problem, or its complexity. Because of this, the *precision* and *recall* of T are also both important. Precision measures how often T derives *true* correctly, in comparison to how often it derives *true* overall. For instance, a precision of 0.75 would indicate that, in 3 out of 4 cases, T answers *true* for query Q when Q is actually true, while in 1 out of 4 cases it incorrectly answers *true* when the proper answer is *false*. A sound analysis will have a precision of 1.0 since it never incorrectly answers *true*. Recall instead measures how often T answers *true* when it could, correctly, answer *true*. For instance, a recall of 0.6 means that, given M and C where Q is *true*, T correctly answers *true* 60% of the time and otherwise answers *false*. A complete analysis has a recall of 1.0. Since computation of both precision and recall require knowledge of the correct answers to Q across the input M and

C, these values would generally be computed as a benchmark across an existing collection of known inputs as a way to test the quality of the analysis.

Note that, along with what has been discussed above, an analysis also needs to satisfy some more basic requirements. For instance, the results of running an analysis should be both repeatable and reproducible. By *repeatable*, we mean that an analysis set up under identical conditions will yield the same results. By *reproducible*, we mean that, given the proper instructions, an analysis set up and conducted by different operators on the same models or systems of interest will yield the same results.

2.2.7 Analysis Composition

It is not always possible to answer a query using a single model or a single property or one analysis technique. There are various reasons for this, such as

(i) The high computational complexity of an analysis technique to handle large and complex models and queries, e.g., model checking of large models is still a major issue due to the state explosion problem,
(ii) The lack of expressiveness of the modelling or property language to express any model or query, for instance a temporal property cannot be expressed using propositional logic, or
(iii) Infeasibility of designing techniques (both, modelling and analysis techniques) to answer all queries, as each technique can be used to answer a specific class of queries.

Different modelling and analysis techniques can be composed to answer a query properly. For instance, to ensure that a large-scale system is trustworthy, several aspects of security and safety should be checked and analysed, where each aspect itself needs different techniques to be analysed, possibly at different levels of abstraction, at different stages of the system life-cycle, with various classes of properties, etc. Security consists of three main aspects, namely confidentiality, integrity, and availability. Confidentiality can concern confidentiality of communications, computations, or storage, and different methods can be used to specify, enforce, and ensure each case. For example, encryption can be used to ensure confidentiality of data during communication while information flow control mechanisms can be used to ensure that computations will not leak information.

We may formulate analysis composition in terms of the following general rule using the judgement introduced earlier in this section:

$$\frac{M_1, C_1 \vdash_{T_1} Q_1 \rightsquigarrow A_1 \qquad M_2, C_2 \vdash_{T_2} Q_2 \rightsquigarrow A_2 \qquad \psi}{M, C \vdash_T Q \rightsquigarrow A}$$

where $M = M_1||_m M_2$ is the composition of models using the composition operator $||_m$, $C = C_1||_c C_2$ is the result of composing the contexts using the operator $||_c$, T is the composed analysis technique, $Q = Q_1 ||_Q Q_2$ is the new query as the result of composing Q_1 and Q_2, and A is the final answer result of the composition of partial answers A_1 and A_2. This rule informally states that if the query Q_i on the model M_i with the context C_i using the analysis technique T_i leads to the answer A_i, $i \in \{1, 2\}$, the query Q using a combination of techniques T_1 and T_2 under the condition ψ will lead to the answer A on the composed model M in the composed context C. The side-condition ψ specifies the conditions under which this composition can be performed, as it is not always possible to arbitrarily compose analyses.

Establishing the composed judgement using this rule can be done in a mathematically sound way, or informally based on some heuristics or expert knowledge. This depends on several factors, such as the existence of a formal definition of the semantics of the prerequisite judgements, the existence of a suitable algorithm or procedure to divide the problem into smaller problems in a sound way (i.e., basic judgements by decomposing the model, context, and query), etc. As an example, the query “Is this system secure?” can be decomposed into three subqueries, each query stating that the system is secure in terms of confidentiality, integrity, and availability. Similarly, these subqueries can be decomposed further into simpler properties, each of which is possibly analysed using a different technique. This means that basic judgements might be established using different methods, such as verification, performance modelling, model-based testing, simulation, penetration testing, etc. The models or properties of the composed judgements could be specified using a multi-view modelling language or an ordinary single-view language.

We proceed by instantiating this general rule with two classic examples: the assume-guarantee verification [AL95] and Hoare logic. In the case of assume-guarantee verification of concurrent systems/programs, let $M, S \models G$ state that a system with the model M will guarantee the relation G (guarantee), if it runs in an environment ensuring the relation S (assumption) on the states. The model is usually specified using a state transition system and the relation is a predicate that specifies some conditions on the transitions (i.e., a pair of states). A simple assume-guarantee verification rule looks like the following:

$$\frac{M_1, S \cup G_2 \models G_1 \qquad M_2, S \cup G_1 \models G_2 \qquad \psi}{M_1 || M_2, S \models G_1 \cup G_2}.$$

The rule informally states that if a module M_i runs in an environment $S \cup G_j$ and guarantees G_i, $i, j \in \{1, 2\}, i \neq j$, then, if the two modules run concurrently in the environment S, they will together guarantee $G_1 \cup G_2$. The notation $\cup$ is used to show the union of two relations. This entailment relation $M, S \models G$ can be expressed as $M, S \vdash_V G \rightsquigarrow \mathsf{True}$ using our above judgement. The model composition operator $||_m$ is a formal well-defined parallel composition operator that computes the product of the two models. The context composition operator returns the intersection of two relations. Such assume-guarantee judgements should be used with caution as they have subtle conditions for validity [AL93].

As the second instantiation, let $\{p\}c\{q\}$ be a Hoare logic’s judgement for sequential programs that informally states that if a program c starts in a state that satisfies the precondition p, if it terminates, the final state will satisfy the post-condition q. The rule for sequential composition of two programs is specified using the following rule:

$$\frac{\{p\}c_1\{q\} \qquad \{q\}c_2\{r\}}{\{p\}c_1;\, c_2\{r\}}.$$

The judgement $\{p\}c\{q\}$, in terms of our judgement, is specified as $c, p \vdash_V q \rightsquigarrow \mathsf{True}$, where the model is the program semantics usually described using a state transition system, the context is the precondition p, the query is the satisfaction of q on termination, and the composition operator of models is the ordinary sequential composition ";". The composition operator on the contexts returns the context of the first judgement.

References

[25011] ISO/IEC 25010. *ISO/IEC 25010:2011, Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models*. 2011.

[AD94] Rajeev Alur and David L. Dill. "A Theory of Timed Automata". In: *Theor. Comput. Sci*. 126.2 (1994), pp. 183–235. https://doi.org/10.1016/0304-3975(94)90010-8.

[AL93] Martin Abadi and Leslie Lamport. "Composing Specifications". In: *ACM Transactions on Programming Languages and Systems* 15.1 (1993).

[AL95] Martin Abadi and Leslie Lamport. "Conjoining Specifications". In: *ACM Trans. Program. Lang. Syst*. 17.3 (1995), pp. 507–534. https://doi.org/10.1145/203095.201069.

[Ald+19] Jonathan Aldrich, David Garlan, Christian Kästner, Claire Le Goues, Anahita Mohseni-Kabir, Ivan Ruchkin, Selva Samuel, Bradley R. Schmerl, Christopher Steven Timperley, Manuela Veloso, Ian Voysey, Joydeep Biswas, Arjun Guha, Jarrett Holtz, Javier Cámara, and Pooyan Jamshidi. "Model-Based Adaptation for Robotics Software". In: IEEE Softw. 36.2 (2019), pp. 83–90. https://doi.org/10.1109/MS.2018.2885058.

[AM11] Musab AlTurki and José Meseguer. "PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool". In: *4th International Conference on Algebra and Coalgebra in Computer Science, CALCO*. Vol. 6859. 2011, pp. 386–392. https://doi.org/10.1007/978-3-642-22944-2_28.

[Amr+15] Moussa Amrani, Benoît Combemale, Levi Lucio, Gehan M. K. Selim, Jürgen Dingel, Yves Le Traon, Hans Vangheluwe, and James R. Cordy. "Formal Verification Techniques for Model Transformations: A Tridimensional Classification". In: *J. Object Technol*. 14.3 (2015), 1:1–43. https://doi.org/10.5381/jot.2015.14.3.a1.

[Bar+09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Vol. 185. 2009, pp. 825–885. https://doi.org/10.3233/978-1-58603-929-5-825.

[Bau+10] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert B. France, Yves Le Traon, and Jean-Marie Mottu. "Barriers to systematic model transformation testing". In: *Commun. ACM* 53.6 (2010), pp. 139–143. https://doi.org/10.1145/1743546.1743583.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. "Modeling Heterogeneous Realtime Components in BIP". In: *Fourth IEEE International Conference on Software Engineering and Formal Methods, SEFM*. 2006, pp. 3–12. https://doi.org/10.1109/SEFM.2006.27.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. "A Tutorial on Uppaal". In: *4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*. 3185. Sept. 2004, pp. 200–236.

[Ben+95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "UPPAAL—a Tool Suite for Automatic Verification of Real–Time Systems". In: *Workshop on Verification and Control of Hybrid Systems III*. 1066. 1995, pp. 232–243.

[Béz+06] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. "Model transformations? transformation models!" In: *International Conference on Model Driven Engineering Languages and Systems*. 2006, pp. 440–453.

[BGS19] Nelly Bencomo, Sebastian Götz, and Hui Song. "Models@run.time: a guided tour of the state of the art and research challenges". In: *Software & Systems Modeling* 18.5 (2019), pp. 3049–3082. https://doi.org/10.1007/s10270-018-00712-x.

[Bib+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. "Model-Driven Development of a Digital Twin for Injection Molding". In: *Advanced Information Systems Engineering*. 2020, pp. 85–100.

[Bin00] Robert V. Binder. *Testing Object-Oriented Systems - Models, Patterns, and Tools*. Addison-Wesley, 2000.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. "The Spec# Programming System: An Overview". In: *International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS*. Vol. 3362. 2005, pp. 49–69.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[Bru+15] Jean-Michel Bruel, Benoit Combemale, Ileana Ober, and Hélène Raynal. "MDE in Practice for Computational Science". In: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS, pp. 660–669. https://doi.org/10.1016/j.procs.2015.05.182.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer, 2001.

[BSA17] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. "A survey on search-based model-driven engineering". In: *Automated Software Engineering* 24.2 (2017), pp. 233–294. https://doi.org/10.1007/s10515-017-0215-4.

[Bur+03] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. "An overview of JML tools and applications". In: *8th International Workshop on Formal Methods for Industrial Critical Systems, FMICS*. Vol. 80. 2003, pp. 75–91. https://doi.org/10.1016/S1571-0661(04)80810-7.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *4th ACM Symposium on Principles of Programming Languages, POPL*. 1977, pp. 238–252. https://doi.org/10.1145/512950.512973.

[CES83] Edmund M. Clarke, E. Allen Emerson, and Aravinda Prasad Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach". In: *Tenth Annual ACM Symposium on Principles of Programming Languages*. 1983, pp. 117–126. https://doi.org/10.1145/567067.567080.

[CGL17] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. "Static Analysis of Model Transformations". In: *IEEE Trans. Software Eng*. 43.9 (2017), pp. 868–897. https://doi.org/10.1109/TSE.2016.2635137.

[CH06] Krzysztof Czarnecki and Simon Helsen. "Feature-based survey of model transformation approaches". In: *IBM Syst. J*. 45.3 (2006), pp. 621–646. https://doi.org/10.1147/sj.453.0621.

[Cla+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Springer, 2007. https://doi.org/10.1007/978-3-540-71999-1.

[Cla+11] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. “Model Checking and the State Explosion Problem”. In: *International Summer School on Tools for Practical Software Verification, LASER*. Vol. 7682. 2011, pp. 1–30. https://doi.org/10.1007/978-3-642-35746-6_1.

[Cla11] Mickael Clavreul. “Model and Metamodel Composition: Separation of Mapping and Interpretation for Unifying Existing Model Composition Techniques”. PhD thesis. University of Rennes 1, France, 2011. https://tel.archives-ouvertes.fr/tel-00646893.

[Com+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, Jim R.H. Steel, and Didier Vojtisek. *Engineering Modeling Languages*. Chapman and Hall/CRC, 2016, p. 398. http://mdebook.irisa.fr/.

[Com+20] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Hyacinth Ali, Daniel Amyot, Mojtaba Bagherzadeh, Edouard Batot, Nelly Bencomo, Benjamin Benni, Jean-Michel Bruel, Jordi Cabot, Betty H C Cheng, Philippe Collet, Gregor Engels, Robert Heinrich, Jean-Marc Jézéquel, Anne Koziolek, Sébastien Mosser, Ralf Reussner, Houari Sahraoui, Rijul Saini, June Sallou, Serge Stinckwich, Eugene Syriani, and Manuel Wimmer. “A Hitchhiker’s Guide to Model-Driven Engineering for Data-Centric Systems”. In: *IEEE Software* (2020).

[CS12] Daniel Calegari and Nora Szasz. “Verification of Model Transformations: A Survey of the State-of-the-Art”. In: *XXXVIII Latin American Computer Conference, CLEI*. Vol. 292. 2012, pp. 5–25. https://doi.org/10.1016/j.entcs.2013.02.002.

[CS13] Daniel Calegari and Nora Szasz. “Institution-Based Semantics for MOF and QVT-Relations”. In: *16th Brazilian Symposium on Formal Methods: Foundations and Applications, SBMF*. Vol. 8195. 2013, pp. 34–50. https://doi.org/10.1007/978-3-642-41071-0_4.

[DDZ08] Jürgen Dingel, Zinovy Diskin, and Alanna Zito. “Understanding and improving UML package merge”. In: *Softw. Syst. Model.* 7.4 (2008), pp. 443–467. https://doi.org/10.1007/s10270-007-0073-9.

[Dur+17] Francisco Durán, Antonio Moreno-Delgado, Fernando Orejas, and Steffen Zschaler. “Amalgamation of domain specific languages with behaviour”. In: *J. Log. Algebraic Methods Program.* 86.1 (2017), pp. 208–235. https://doi.org/10.1016/j.jlamp.2015.09.005.

[Dur14] Francisco Durán. “Composition of Graph-Transformation-Based DSL Definitions by Amalgamation”. In: *10th International Workshop on Rewriting Logic and Its Applications, WRLA, Revised Selected Papers*. Vol. 8663. 2014, pp. 1–20. https://doi.org/10.1007/978-3-319-12904-4_1.

[DY83] Danny Dolev and Andrew Chi-Chih Yao. “On the Security of Public Key Protocols”. In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–207. https://doi.org/10.1109/TIT.1983.1056650.

[GL12] Esther Guerra and Juan de Lara. “An Algebraic Semantics for QVT-Relations Check-only Transformations”. In: *Fundam. Informaticae* 114.1 (2012), pp. 73–101. https://doi.org/10.3233/FI-2011-618.

[GL16] Philip Gerlee and Torbjörn Lundh. *Scientific Models*. Springer, 2016.

[GMM19] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. “Automatic Software Repair: A Survey”. In: *IEEE Trans. Software Eng.* 45.1 (2019), pp. 34–67. https://doi.org/10.1109/TSE.2017.2755013.

[Gou+12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. “GenProg: A Generic Method for Automatic Software Repair”. In: *IEEE Trans. Software Eng.* 38.1 (2012), pp. 54–72. https://doi.org/10.1109/TSE.2011.104.

[Gra] GrammaTech. *CodeSonar: Static Code Analysis*. https://www.grammatech.com/products/source-code-analysis (visited on 04/24/2021).

[Gro06] Object Management Group. *MOF Specification Version 2.0*. http://www.omg.org/docs/ptc/06-05-04.pdf. Jan. 2006.

[Gro12] Object Management Group. *OMG Systems Modeling Language (OMG SysML), Version 1.3*. 2012. http://www.omg.org/spec/SysML/1.3/

[GS02] David Garlan and Bradley R. Schmerl. “Model-based adaptation for self-healing systems”. In: *First Workshop on Self-Healing Systems, WOSS*. 2002, pp. 27–32. https://doi.org/10.1145/582128.582134.

[GS18] *International Summer School on Bidirectional Transformations, Tutorial Lectures*. Vol. 9715. 2018. https://doi.org/10.1007/978-3-319-79108-1.

[Hei+17] Robert Heinrich, Reiner Jung, Christian Zirkelbach, Wilhelm Hasselbring, and Ralf Reussner. “Software Architecture for Big Data and the Cloud”. In: 2017. Chap. An Architectural Model-Based Approach to Quality-aware DevOps in Cloud Applications, pp. 69–89.

[Hei+21] Robert Heinrich, Francisco Durán, Carolyn L. Talcott, and Steffen Zschaler (eds.) *Composing Model-Based Analysis Tools*. Springer, 2021. https://doi.org/10.1007/978-3-030-81915-6.

[Hid+16] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. “Feature-based classification of bidirectional transformation approaches”. In: *Softw. Syst. Model*. 15.3 (2016), pp. 907–928. https://doi.org/10.1007/s10270-014-0450-0.

[HKB09] Tingting Han, Joost-Pieter Katoen, and Damman Berteun. “Counterexample Generation in Probabilistic Model Checking”. In: *IEEE Transactions on Software Engineering* 35.2 (2009), pp. 241–257.

[HMZ12] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. “Search-based software engineering: Trends, techniques and applications”. In: *ACM Comput. Surv.* 45.1 (2012), 11:1–11:61. https://doi.org/10.1145/2379776.2379787.

[Hoa69] Charles A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580. https://doi.org/10.1145/363235.363259.

[How77] William E. Howden. “Symbolic Testing and the DISSECT Symbolic Evaluation System”. In: *IEEE Trans. Software Eng*. 3.4 (1977), pp. 266–278. https://doi.org/10.1109/TSE.1977.231144.

[HR04] David Harel and Bernhard Rumpe. “Meaningful modeling: what’s the semantics of “semantics”?” In: *Computer* 37.10 (2004), pp. 64–72.

[Hu+11] Zhenjiang Hu, Andy Schürr, Perdita Stevens, and James F. Terwilliger. “Bidirectional Transformation “bx” (Dagstuhl Seminar 11031)”. In: *Dagstuhl Reports* 1.1 (2011), pp. 42–67. https://doi.org/10.4230/DagRep.1.1.42.

[Joh+19] Stefan John, Alexandru Burdusel, Robert Bill, Daniel Strüber, Gabriele Taentzer, Steffen Zschaler, and Manuel Wimmer. “Searching for Optimal Models: Comparing Two Encoding Approaches”. In: *12th Int’l Conf. on Model Transformations, ICMT*. 2019.

[Jou+08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. “ATL: A model transformation tool”. In: *Sci. Comput. Program*. 72.1-2 (2008), pp. 31–39. https://doi.org/10.1016/j.scico.2007.08.002.

[Kah+19] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Dániel Varró. “Survey and classification of model transformation tools”. In: *Software and Systems Modelling* 18.4 (2019), pp. 2361–2397. https://doi.org/10.1007/s10270-018-0665-6.

[Kah87] Gilles Kahn. “Natural Semantics”. In: *4th Annual Symposium on Theoretical Aspects of Computer Science, STACS*. Vol. 247. 1987, pp. 22–39.

[Kin76] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (1976), pp. 385–394. https://doi.org/10.1145/360248.360252.

[KKS19] Narges Khakpour, Jetty Kleijn, and Marjan Sirjani. “A Formal Model to Integrate Behavioral and Structural Adaptations in Self-adaptive Systems”. In: *8th International Conference on Fundamentals of Software Engineering, FSEN*. Vol. 11761. 2019, pp. 3–19. https://doi.org/10.1007/978-3-030-31517-7_1.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages using Metamodels*. Pearson Education, 2008.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems”. In: *23rd International Conference on Computer Aided Verification, CAV*. Vol. 6806. 2011, pp. 585–591. https://doi.org/10.1007/978-3-642-22110-1_47.

[KS18] Narges Khakpour and Charilaos Skandylas. “Synthesis of a Permissive Security Monitor”. In: *23rd European Symposium on Research in Computer Security, ESORICS, Part I*. Vol. 11098. 2018, pp. 48–65. https://doi.org/10.1007/978-3-319-99073-6_3.

[Küh16] Thomas Kühne. “Unifying Explanatory and Constructive Modeling: Towards Removing the Gulf between Ontologies and Conceptual Models”. In: *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 2016, pp. 95–102. https://doi.org/10.1145/2976767.2976770.

[Kus+17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. “Modeling Architectures of Cyber-Physical Systems”. In: *13th European Conference on Modelling Foundations and Applications, ECMFA*. Vol. 10376. 2017, pp. 34–50. https://doi.org/10.1007/978-3-319-61482-3_3.

[Lee18] Edward A. Lee. “Modeling in Engineering and Science”. In: *Commun. ACM* 62.1 (2018), pp. 35–36. https://doi.org/10.1145/3231590.

[LG10] Juan de Lara and Esther Guerra. “Generic Meta-modelling with Concepts, Templates and Mixin Layers”. In: *13th International Conference on Model Driven Engineering Languages and Systems, MODELS, Proceedings, Part I*. Vol. 6394. 2010, pp. 16–30. https://doi.org/10.1007/978-3-642-16145-2_2.

[LS09] Martin Leucker and Christian Schallhart. “A brief account of runtime verification”. In: *J. Log. Algebraic Methods Program*. 78.5 (2009), pp. 293–303. https://doi.org/10.1016/j.jlap.2008.08.004.

[LV03] Edward A. Lee and Pravin Varaiya. *Structure and interpretation of signals and systems*. Addison-Wesley, 2003.

[Men13] Tom Mens. “Model Transformation: A Survey of the State of the Art”. In: *Model-Driven Engineering for Distributed Real-Time Systems*. 2013. Chap. 1, pp. 1–19. https://doi.org/10.1002/9781118558096.ch1.

[Mes12] José Meseguer. “Twenty years of rewriting logic”. In: *J. Algebraic and Logic Programming* 81 (2012), pp. 721–781.

[Mes92] José Meseguer. “Conditional Rewriting Logic as a Unified Model of Concurrency”. In: *Theoretical Computer Science* 96.1 (1992), pp. 73–155.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice Hall, 1989.

[MR04] José Meseguer and Grigore Rosu. “Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools”. In: *Second International Joint Conference on Automated Reasoning, IJCAR*. Vol. 3097. 2004, pp. 1–44. https://doi.org/10.1007/978-3-540-25984-8_1.

[Mus+09] Mohamed Mussa, Samir Ouchani, Waseem Al Sammane, and Abdelwahab Hamou-Lhadj. “A Survey of Model-Driven Testing Techniques”. In: *Ninth International Conference on Quality Software, QSIC*. 2009, pp. 167–172. https://doi.org/10.1109/QSIC.2009.

[MYR16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. “Angelix: scalable multiline program patch synthesis via symbolic analysis”. In: *38th International Conference on Software Engineering, ICSE*. 2016, pp. 691–701. https://doi.org/10.1145/2884781.2884807.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[OMG16] Object Management Group. *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. 2017. https://www.omg.org/spec/UML/2.5.1/.

[ÖM07] Peter C. Ölveczky and José Meseguer. “Semantics and Pragmatics of Real-Time Maude”. In: *Higher-Order and Symbolic Computation* 20.1-2 (2007), pp. 161–196.

[Obj17] OMG - Object Management Group. MOF Query/View/Transformation. Version 1.3. 2016. https://www.omg.org/spec/QVT/.

[OS08] Sam Owre and Natarajan Shankar. "A Brief Overview of PVS". In: *21st International Conference on Theorem Proving in Higher Order Logics, TPHOLs, Proceedings*. Vol. 5170. 2008, pp. 22–27. https://doi.org/10.1007/978-3-540-71067-7_5.

[Plo04] Gordon D. Plotkin. "A structural approach to operational semantics". In: *Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 17–139.

[REP12] Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. "Model Transformations". In: *12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM*. Vol. 7320. 2012, pp. 91–136. https://doi.org/10.1007/978-3-642-30982-3_4.

[RES10] Grigore Rosu, Chucky Ellison, and Wolfram Schulte. "Matching Logic: An Alternative to Hoare/Floyd Logic". In: *13th International Conference on Algebraic Methodology and Software Technology, AMAST*. Vol. 6486. 2010, pp. 142–162. https://doi.org/10.1007/978-3-642-17796-5_9.

[Reu+16] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016.

[RG00] Mark Richters and Martin Gogolla. "Validating UML Models and OCL Constraints". In: «UML» *2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference*. Vol. 1939. 2000, pp. 265–277. https://doi.org/10.1007/3-540-40011-7_19.

[RN08] Arend Rensink and Ronald Nederpel. "Graph Transformation Semantics for a QVT Language". In: *Electron. Notes Theor. Comput. Sci. 211* (2008), pp. 51–62. https://doi.org/10.1016/j.entcs.2008.04.029.

[Roz97] *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[RW15] Lukman Ab. Rahim and Jon Whittle. "A survey of approaches for verifying model transformations". In: *Softw. Syst. Model*. 14.2 (2015), pp. 1003–1028. https://doi.org/10.1007/s10270-013-0358-0.

[SCD12] Gehan M. K. Selim, James R. Cordy, and Juergen Dingel. "Model transformation testing: the state of the art". In: *First Workshop on the Analysis of Model Transformations, AMT*. 2012, pp. 21–26. https://doi.org/10.1145/2432497.2432502.

[SGW94] Bran Selic, Garth Gulkeson, and Paul Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[Sto77] Joseph E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory*. MIT Press, 1977.

[Syn] Synopsys. *Coverity Scan Static Analysis*. https://scan.coverity.com/ (visited on 04/24/2021).

[Tom99] Paul Tomassi. "An Introduction to First Order Predicate Logic". In: 1999, pp. 205–280.

[TV10] Javier Troya and Antonio Vallecillo. "Towards a Rewriting Logic Semantics for ATL". In: *3rd International Conference on Theory and Practice of Model Transformations, ICMT*. Vol. 6142. 2010, pp. 230–244. https://doi.org/10.1007/978-3-642-13688-7_16.

[TV11] Javier Troya and Antonio Vallecillo. "A Rewriting Logic Semantics for ATL". In: *J. Object Technol*. 10 (2011), 5: 1–29. https://doi.org/10.5381/jot.2011.10.1.a5.

[Voe+13] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schätz. "mbeddr: instantiating a language workbench in the embedded software domain". In: *Automated Software Engineering* 20.3 (2013), pp. 339–390. https://doi.org/10.1007/s10515-013-0120-4.

[WTZ10] Christian Wende, Nils Thieme, and Steffen Zschaler. "A Role-based Approach Towards Modular Language Engineering". In: *2nd Int'l Conf. on Software Language Engineering*, SLE. Vol. 5969. 2010, pp. 254–273.

[ZP20] Steffen Zschaler and Fiona Polack. “A Family of Languages for Trustworthy Agent-Based Simulation”. In: *13th International Conference on Software Language Engineering, SLE*. 2020.

[Zsc09] Steffen Zschaler. “Formal Specification of Non-functional Properties of Component-Based Software Systems: A Semantic Framework and Some Applications Thereof”. In: *Software and Systems Modelling* 9.2 (2009), pp. 161–201. https://doi.org/10.1007/s10270-009-0115-6.