CHAPTER 3

CONSIDERATIONS AND RATIONALE FOR A UML SYSTEM MODEL

Manfred Broy[1] and María Victoria Cengarle[1] and
Hans Grönniger[2] and Bernhard Rumpe[2]

[1]Software and Systems Engineering, Technische Universität München, Germany
[2]Software Systems Engineering, Technische Universität Braunschweig, Germany

3.1 INTRODUCTION

Semantics definition for the Unified Modeling Language (UML) [33, 8] is not an easy task. Although considerable efforts have been made starting in the late nineties [19], no commonly agreed, formal and integrated semantics of the UML exists. In [1], we have defined a system model as a semantic domain for the UML. The system model is supposed to form a possible core and foundation of the UML semantics definition. For that purpose, the definitions are targeted towards UML which means that central concepts of UML have been formalized as theories of the system model.

This contribution is structured as follows: In the rest of this Chapter, we discuss the general approach and highlight the main decisions. This Chapter is important to understand the system model definition, given in Chapter 4. This work is based on

UML2 Semantics and Applications. By Kevin Lano

the second version of the system model [1] which is the result of a major effort to define the structure, behaviour and interaction of object-oriented, possibly distributed systems abstract enough to be of general value, but also in sufficient detail for a semantic foundation of the UML. The first version of the system model can be found in [4, 5, 6].

3.2 GENERAL APPROACH TO SEMANTICS

The semantics of any formal language consists of the following basic parts [44]:

- the syntax of the language in question (here: UML) – be it graphical or textual,
- the semantic domain, a domain well-known and understood based on a well-defined mathematic theory, and
- the semantic mapping: a functional or relational definition that connects both, the elements of the syntax and the elements of the semantic domain.

This technique of giving meaning to a language is the basic principle of denotational semantics: every syntactic construct is mapped onto a semantic construct. As discussed in the literature, there are many flavors of these three elements. Syntax can, for example, be specified by grammars or metamodels. To stay formal, our approach intends to use the abstract syntax of UML in a mathematical form that resembles context-free grammars, examples are given in [11, 10]. In [24] the term system model was used the first time to denominate a semantic domain; it defines a family of systems, describing their structural and behavioural issues. Each concrete syntactic instance (in our case, an individual UML diagram, or even a part of it) is interpreted by the semantic mapping as a predicate over the set of systems defined by the system model. As explained in [21] the semantic mapping has the form:

$$Sem : UML \rightarrow \mathbb{P}(Systemmodel)$$

and thus functionally relates any item in the syntactic domain to a set of constructs of the semantic domain. The semantics of a model $m \in UML$ is therefore $Sem(m)$.

Given any two models $m, n \in UML$ combined into a complex one $m \oplus n$ (for any composition operator $\oplus$ of the syntactic domain), the semantics of $m \oplus n$ is defined by $Sem(m \oplus n) = Sem(m) \cap Sem(n)$. This definition also works for sets of UML documents which allows an easy treatment of views on a system specified by multiple UML diagrams. The semantics of several views, i.e., several UML documents is given as $Sem(\{doc_1, \ldots, doc_n\}) = Sem(doc_1) \cap \ldots \cap Sem(doc_n)$. A set of UML models *docs* is consistent if systems exist that are described by the models, so $Sem(docs) \neq \varnothing$. As a consequence, the system model supports both view integration and model consistency verification.

In the same way, $n \in UML$ is a (structural or behavioural) refinement of $m \in UML$, exactly if $Sem(n) \subseteq Sem(m)$. Formally, refinement is nothing else than "n is providing at least the information about the system that m does". These

general mechanisms provide a great advantage, as they simplify any reasoning about composition and refinement operators.

The system model described in this document identifies the set of all possible object-oriented (OO) systems that can be defined using a subset of UML which we call "clean UML" as introduced below. It relies on earlier work on system models [9, 24, 20, 2, 1, 40].

To capture and integrate all the orthogonal aspects of a system modeled in UML, the semantic domain necessarily has to have a certain complexity. Related approaches very often contain a relatively small and specialized semantic domain, such as (pairs of) sets of traces for UML interaction [22], template semantics based on hierarchical state machines [42] or Kripke structures [43] for UML State Machines, or sets of inequations to give semantics to class diagrams focusing on satisfyability of association cardinality [38, 8, 30, 18]. However, these approaches fail to give an integrated semantics for different types of UML notations. Approaches with a broader scope are for example [14] which define a UML subset called krtUML and associates with each model a symbolic transition system. [27] combine class, object and state machine diagrams using graph transformations. In [15] dynamic metamodeling (also based on graph transformations) is used to define the operational semantics of, e.g., UML activities. Semantics for class and state machine diagrams have been developed for different purposes. [39] examines the refinement of associations. [17] provide a compositional semantics that considers activity groups. [28] additionally supports sequence diagrams and considers timing issues. In [45] consistency between (simplified) state machines and sequence diagrams is checked using a model checker. Consistency conditions are also proposed [29, 35].

3.3 STRUCTURING THE SEMANTICS OF UML

Our long term goal is to define the semantics of a comprehensive core of well-defined concepts of UML.

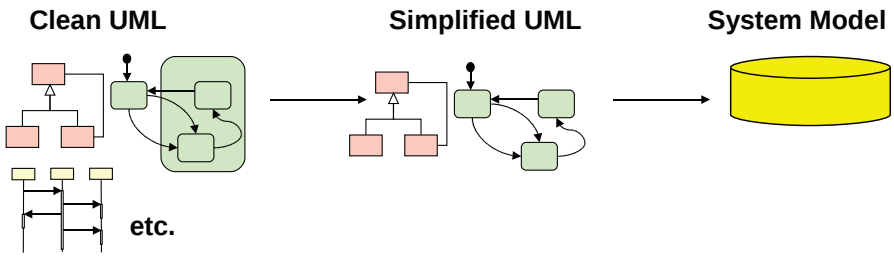

Figure 3.1 General strategy for the definition of the semantics of UML 2.0

The overall strategy of giving semantics to a modeling language is depicted in Figure 3.1. The basic idea expressed by this diagram is as follows:

- Full UML is restricted to a subset (called "clean UML") that can be treated semantically without overly sophisticated constructs.
- Clean UML is mapped by transformations into Simplified UML. In doing so, derived constructs of UML are replaced by their definition in terms of constructs of the core. That way, notational extensions and derived concepts can be eliminated. UML provides a number of derived operators which do not enhance the expressiveness of the language but the comfort of its use. Derived constructs can be defined in terms of constructs of the core as, e.g., state hierarchy of UML's state transition diagrams can be neglected without losing expressiveness.
- Simplified UML, finally, is mapped to the system model using a predicative approach.

The system model describes the "universe (set) of all possible semantic structures (each with its behaviour)". The semantic mapping interprets a UML model as a predicate that restricts the universe to a certain set of structures, which represents the meaning of the UML model. To be able to faithfully map concepts from UML to the system model, the system model has to cover a number of basic concepts expressible in UML. Otherwise, the semantic mapping cannot be defined in an adequate manner.

The system model itself is defined in a modular fashion. From a global viewpoint, a system in the system model is a state transition system. This semantic universe is introduced in layers of mathematical theories which are shown in Figure 3.2. Please note that this figure shows the full set of theories as defined in [1], in this text we slightly shorten the definitions.

The rectangles in Figure 3.2 contain names of the theories, whereas arrows show a relationship among concepts that could be paraphrased as "is defined in terms of". For instance, basic theories for types and objects are used to define the data, control, and event state of a system, that in turn are used to define the state space for the transition systems.

When defining the constituents of the system model, we will state the decisions that have to be made, that can be left open or do not even occur when staying informal. We clearly identify those decisions either directly, or mark them as a "variation point" and leave it to the user of the system model to choose or adopt a variation. Those variation points may very nicely correspond to stereotypes on the language side, such that the language designer (and semantics definer) can transfer the freedom of choice to the actual modeler.

3.4 THE MATH BEHIND THE SYSTEM MODEL

A precise description of the system model calls for a precise instrument. For our purposes, mathematics is exactly appropriate because of its power and flexibility. Admittedly, reading and understanding mathematics is an effort that requires some training, but it allows for precisely and abstractly describing things that cannot be

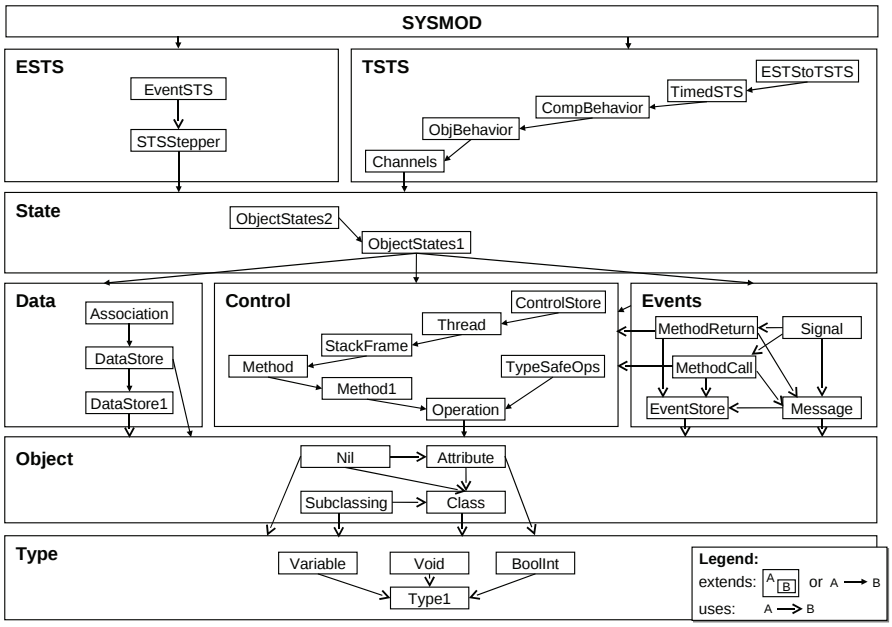

Figure 3.2 Theories that constitute the system model

defined using, e.g., UML itself. Using UML itself to describe semantics of UML might seem, a pragmatic approach. This approach, however, is somewhat meta-circular and necessarily calls for a kind of bootstrap, typically mathematics again. Moreover, understanding the semantics of UML in terms of UML itself demands a very good knowledge of the language whose semantics is about to be formally given. Besides, UML does not conveniently provide the appropriate mechanisms we need, e.g., to handle scheduling, distributed systems and to deal with underspecification in a precisely controllable way. Of course, whenever appropriate, we use diagrams to illustrate some mathematically defined concepts, but the diagrams do not replace the mathematical formulas.

Instead of relying on basic mathematics, related work often proposes the use of specialized formalisms. [7, 16] translate UML to the formal language Z while [37] map to B. Graph transformations are used in [27]. The process algebra π-calculus has been proposed to model activities [26] that also have been formalised using Petri nets [41], or Abstract State Machines (ASMs) [37]. Trace-based semantics for interactions have been presented in [22, 12]. Metamodeling techniques have been employed by [15]. Template semantics [42] that are based on state machines allow for describing semantic variation points.

We intentionally avoided the use of more specialized notations such as Z, B, ASMs, etc. for two reasons.

- It is not clear that any of these notations is general and comfortable enough to allow a satisfactory and adequate expression of all concepts in UML.
- Arguably, all these notations have a certain bias (e.g., for state-based formal specification, analysis with a theorem prover, analysis with a model checker); we kept the system model free of this bias to ensure that we obtain a true reference semantics that, if useful, enables the future use of other notations for, e.g., analysis purposes.

Because of these reasons we decided to use only mathematics. The following principles have proven to be useful when defining the system model:

1. Mathematics is used to define the system model. Its sub-theories are built on: numbers, sets, relations, and functions. Additional theories are built in a layered form. That is, only notation and mathematical definitions and neither new syntax nor language are introduced or used in the system model. Diagrams are occasionally used to clarify things, but do not formally contribute to the system model.
2. The system model does not constructively define its elements, but introduces the elements and characterizes their properties. That is, abstract terms are used whenever possible. For instance, instead of using a record to define the structure of an object, we introduce an abstract set of objects and a number of selector functions. Properties of the set are then defined through such selectors. Based on our background and knowledge, we claim that we can transform this system model into a constructive version (and actually do this, cf. [9]), but that would probably be more awkward to read and less intuitive, as it costs a lot more mathematical machinery. This will satisfy "constructivists" who wish everything being constructive or executable.
3. Everything important is given an appropriate name. For instance, in order to deal with classes, there is a "universe of class names" UCLASS, and similarly there is also a "universe of type names" UTYPE, which however is just a set of names (and not types); see Sects. 4.3.4 and 4.3.1 below.
4. To our best knowledge, any underlying assumptions were avoided, according to the slogan: What is not explicitly specified does not need to hold. If we, for instance, do not explicitly state that two sets are disjoint, these two sets might have elements in common. Sometimes these loose (underspecified) ends are helpful to specialize or strengthen the system model and are there on purpose. If you need a property, (a) check whether it is there, (b) if absent, check whether it can be inferred as an emerging property, (c) if not, check if it is absolutely necessary, and (d), if yes, you may add it as an additional restriction.
5. Generally, deep embedding (or explicit representation) is used. This means the semantics of the embedded language, i.e., UML, is completely formalized within the supporting language, in our case, mathematics. As one consequence,

although there are similar concepts in the language describing the system model (which is mathematics) and the language described (UML), these need not be related. For instance, the system model characterizes the type system of UML, it however does not have and does not need a type system itself.

6. Specific points, where the system model could be further strengthened, have been marked as "variation points". Variation points deal with additional elements that can be defined upon the system model. We may introduce additional machinery that needs not be present in each modeled system. Prominent examples of such variations are the existence of a predefined top-level class called "Object" or an enhanced type system, including, e.g., templates. Furthermore, variations describe changes of definitions, that lead to a slightly different system model. Variation points allow us to describe specialized variants of the system model, that may not be generally valid, but hold for a large part of possible systems. Examples are single inheritance hierarchies or type-safe overriding of operations in subclasses, which may not be assumed in general.

3.5 WHAT IS THE SYSTEM MODEL?

As already indicated in Sect. 3.3, the system model is a hierarchy of theories that capture a large number of concepts typically found in distributed object-oriented systems. To obtain an adequate semantic domain, the system model defines concepts such as types and values, classes with attributes and methods, objects, messages and events, or threads.

An object-oriented system can basically be described using one of various existing paradigms. We opted for the paradigm of a global state transition system in order to accommodate a global (and maybe distributed) state space. The system model, thus, defines a universe of state transition systems. A state transition system is given by its state space, its initial states, and its state transition function. Note that our notion of state transition system is more basic and does not directly relate to the state machines the UML provides. The global state transition system, if detailed enough, is perfectly appropriate to model parallel, independent and distributed computations. In principle, a system of communicating, elementary transition systems could be considered more convenient than a single, global machine for describing the semantics of UML models. It is also possible to construct a global transition system by integrating elementary ones; however, this is a non-trivial operation. Therefore, it is more appropriate to employ the concept/metaphor of one state transition system at a higher, non-elementary level. In fact, we introduce a composition operator on transition systems representing fragments of larger systems, such that these transition system can be composed, leading to larger systems.

Static and Dynamic Issues

The types and classes are static, i.e., they do not change over the lifetime of a system. Similarly, the sets of defined operations, methods, messages, and events do not change. This information is called the static information of a state transition system.

The set of existing objects, the values of the attributes, the computational state of invoked methods, and dispatched and not yet delivered messages passed from one object to another one are dynamic, i.e., they may change in transition steps. This is called the dynamic information of a state transition system and is encoded in the states of the system. In the database realm, the static part is called "schema", and the dynamic part is the "instance". The schema instantiation is changeable while the schema itself is not. Schema changes (usually called "schema evolution" in the literature) are not considered, as they usually do not occur within a running system, but when evolving and/or reconfiguring it.

Summarizing, the state space of the transition system will be defined in terms of the orthogonal constituents data, control, and events. Each of these theories contributes static and dynamic information to the system model definitions.

Types, Classes, Objects, and DataStore

The first part of the system model definition (c.f. Sect. 4.3) is concerned with defining type names, their carrier sets, classes, objects, associations, and the component of a system state that stores information about existing objects and their variable values.

Although we do not deal with peculiarities of various type systems, strong or weak typing, etc., we outline basic assumptions on the underlying type system, as we need to map the type information of UML to this type system. In that respect, we use a deep embedding of the type system of UML, by representing it through type names and a universe of values only. By deep embedding, we mean that we do not map types of the UML to a type system of the underlying mathematical structure, but explicitly model types as first-class elements.

Occasionally, we make assumptions that simplify matters but pay attention that we do not lose generality. For example, we assume global variables in the system. In practice, it would be relatively inconvenient if every variable name could only be used once in a program. We then would see a global namespace and thus not have any hiding concepts in the language. In the system model, however, we may accept such a restriction, and handle it as follows. Like in ordinary programming languages, variables shadow each other when a new variable with the same name is introduced in an inner scope. We assume static binding, thus each variable name can be statically resolved (as opposed to dynamic binding of variables by which the resolution of a variable name depends not on the environment of its definition but on the environment of its use, and thus variable resolution can only occur at run time). Generally, we assume that in the modeling languages we deal with, a consistent and model-wide redefinition of variable names is possible in such a way that each variable is used only once. Then variable shadowing does not occur and any variable is unique. We may

handle that systematically through encoding the place of definition or the namespace within each variable. Quite the same is done by many compilers anyway.

Class names in the system model are introduced in an abstract fashion. Each class name is associated with a set of object identifiers and with a set of attributes. This is sufficient to define the structure of objects belonging to a class in form of a tuple, consisting of object identifier and a mapping of attributes to values.

In the system model, classes are also types. Together with a subclassing relation, the carrier set of a class is the set of object identifiers belonging to the class or to one of its subclasses. This allows to polymorphically store subclass identifiers in places where superclass identifiers are expected. As a consequence, we require object identifiers to be values. However, objects will not be forced to be values. We leave open whether objects are also to be treated as values (variation point). Our relational point of view concerning subclassing also supports multiple inheritance which is covered by several binary inheritance relationships. As we assume global attribute names, we avoid name conflicts that otherwise could arise.

For the data store, we abstract from a number of details, such as storage layout and physical distribution. We use an abstract global store to denote the data state of an object system. Even if there is no such concept in the real, possibly distributed system, we can conceptually model the system that way by organizing all instances in this single global store. We also allow interleaving, as well as concurrent activities, as can be seen in the control part of the system model in Section 4.4.

Intuitively, the data store models the data state of a system at a certain point in time. Normally, at each point of time the store contains real objects for a finite subset of the universe of all object identifiers. We will, however, see that the data store is not enough to describe the system, but a control store and an event store need to be added. In these stores time progress is modeled by state transitions of the overall state machine.

At each point in time, i.e., in each state of the state machine, when an instance exists, we assume that its attributes are present and their values are defined, but it is not necessarily the case that we do know about these values. They may be left underspecified. In particular it may be that, after creation of an instance, its attributes still need to be initialized, i.e., come into a known (and thus well-defined) state. Note that this is a usual modeling technique used, e.g., in verification systems to avoid an explicit handling of a pseudo-value “undefined” [32]. It also resembles reality, e.g., when an uninitialized variable of type *int* is accessed, we do know that it contains an integer, but we do not have a clue which one it is.

One of the core concepts of UML are associations. Associations are relations between classes; and links, which can be regarded as instances of associations, are the corresponding relations between object identifiers at runtime. While associations are mostly binary, they may be of any arity, in addition they may be qualified in various ways and may have additional attributes on their own. Furthermore, an association can be “owned” by one or more of the participating objects/classes or can stand on its own, not owned by any of the related objects. In an implementation a basic mechanism for managing those relations is to use direct links or Collection classes but there are other possibilities as well. To semantically capture different

variants of realizations of associations, we use a generalized, extensible approach: Retrieval functions extract links from the store. We allow for a variety of realizations of these functions. This approach is very flexible as it, on the one hand, abstracts away from the owner of associations as well as from how associations are stored and, on the other hand, does not restrict possible forms of an association. As a big disadvantage of this approach, we cannot capture all forms of associations in one uniform characterization, but need to provide a number of standard patterns that cover the most important cases. If no standard case applies, e.g., for a new stereotype for associations, then the stereotype developer has to describe his/her interpretation of the stereotype directly in the terms of the system model. We demonstrate this approach by defining variants of binary associations below.

The retrieval function *relOf* depends on the concrete realization of the association. Even after quite a number of years of studying formalizations of object orientation, there is so far not a really satisfactory approach describing all variants of association implementations. Therefore, we provide this abstract function and impose some properties on the function without discussing the internal storage structure. The only decision we made so far is that associations are somehow contained within the store, i.e., they are somehow part of objects and association relations do not extend the store. This is pretty much in the spirit of the system model where higher-level concepts are explained using lower-level concepts. In order to retrieve the links of an association, the state of multiple objects may have to be examined. From the viewpoint of a single object, this is not possible since it only has access to its own state. Hence, we assume that links may be retrieved using an "API", i.e., special methods that can be called by an object and that return the links.

Operation, Methods, Threads, and ControlStore

The control part defines the constituents of the structure used to model control information like operations and methods. The control store contains additional information needed to determine the state of the system during computation. In particular, we provide means to express how control flows (as part of method calls) through active and passive objects, what it means for an object to be active or passive, how messages are passed, delayed and handled, how events are handled, how threads work in a distributed setting, and how synchronization of all these concepts takes place, c.f. Sect. 4.4.

One result of this section is a flexible mechanism to describe control structures of various kinds resembling quite a number of implementation languages. This variability is enforced by the UML and leads to a rather complex formalization of control. In fact, UML does not allow us to abstract away from control primitives. In the systems we describe with UML, we do not only have various types of control and interaction, but also very often their combinations within a single system. Unfortunately, we need rather detailed definitions for stacks, events, and threads that are not very elegant and do not give us much abstraction. However, this lack of elegance accurately covers the lack of elegance in distributed object-oriented systems where method calls, asynchronous signals and threads of activity are orthogonal concepts that can be mixed

in various ways. On the one hand, these concepts provide the system developer with great flexibility. On the other hand, they make it difficult to understand the behaviour of the resulting systems. In addition, many orthogonal concepts make it very awkward to describe a system model that uses all of them, because any combination (useful or not) needs to be covered. The resulting complexity becomes apparent in modeling the control part of the system model.

We define operations (signatures) and methods (implementations) as separate entities. Operations are named, have a list of parameter types and one return type. Methods additionally define parameter names and may have an implementation. This approach does not explicitly specify overloading, signature and implementation inheritance, overriding and dynamic binding but allows specializations in a flexible way to various actually used mechanisms of method binding. This even includes binding mechanisms such as that in Modula-3. These concepts, thus, are to be decided and defined by the time the mapping from UML to the system model is devised.

In UML, interestingly, subclassing does not impose clear constraints on method implementations, as the implementation may be redefined according to some "compatibility" notion. This notion, however, is a semantic variation point that we therefore also leave open to a semantic specialization, e.g., by adding additional constraints for redefined method behaviour. Subclassing in general allows for renaming of parameters in the implementation, as those are not part of the signature. The signatures (in the form of lists of types), however, are either equal or in a generalization/specialization relation. The types of parameters can be generalized, and the type of the return value can be specialized. This is the well-known co/contra-variant way (see, e.g., [7]) that ensures type safety in a language. We also impose this constraint in the system model.

UML furthermore provides "out" and "in/out" parameters. Many authors however advise against the use of (in/)out parameters. The recommendation in the present context is to use a variation point where, if several "out"-values are to be assigned, each of these is assigned through method call or message passing. In this way, object encapsulation is kept. However, if needed, the system model allows to encode these parameters by passing locations of the variables where the "out"-values are to be stored.

In UML there is also the notion of "object behaviour", which, strictly speaking, is not a method. However, for simplification we assume that "object behaviour" can be encoded as a special kind of operation associated with the object whose parameters define the signature of the operation.

The computational state of a method is stored in a frame with the obvious information like sender, receiver, values of local variables and parameters. A thread in the system model is associated with a stack of frames. The control store is the part of a system model state that information about which threads currently execute methods in which objects.

There are quite a number of approaches to combine object orientation and concurrency. Some approaches argue that each object is a unit of concurrency on its own. Others group passive objects into regions around single active objects, allowing

operation calls only within a region and message passing only between regions. The programming languages that are commonly used today, however, have concurrency concepts that are completely orthogonal to objects. This means, various concurrent threads may independently and even simultaneously "enter" the very same object. The system model is abstract enough to allow specializations to any of these approaches. We do, however, have the basic assumption that there is a notion of atomic action. These atomic actions are the basic units for concurrency; their exact definition is deferred to the UML actions definitions. On top of atomic actions we assume forms of concurrency control that are provided through appropriate concepts in UML (like "synchronized" in Java). However, UML currently does not provide sufficient mechanisms to actually define scheduling and atomicity of actions conveniently. Possible units of concurrency, for example, would be a variable assignment or an operation invocation.

Messages, Events, and EventStore

One crucial question is the choice of the appropriate communication or interaction mechanism. Two basic flavors are asynchronous and synchronous communication. There is no definitive answer as to which one of these two possibilities is better, and both approaches can model each other. The system model is based on the asynchronous approach because of its abstractness. Synchronous method calls within the system model are encoded as asynchronous message passing.[1]

The UML specification distinguishes between event (types) and event occurrences (cf. [8, Sect. 6.4.2]) and provides a rather general notion of events and event occurrences. An event may be a message (which resembles a method call with parameters or return values), a *timeout*, a simple *signal*, or a *spontaneous state change*. Event occurrences, for example, are *sending of a message* or *reception of a message*. In Sect. 4.5.1 we introduce events, and subsets of events that contain messages (which may further contain method calls and returns) and signals.

The last constituent of the state of an object is the event store. For each existing object, the event store stores a buffer that contains the events that still need to be processed. Event occurrences correspond to system states in which an event has just been added (sent) or removed (received) from the event store.

States

The state of a system is straightforwardly defined to consist of one data store, one control store, and one event store (see Sect. 4.6). So in each system state we capture the attribute values, the computational state, and the event buffer of each object.

Given a system state, the state of an individual object is consequently the part of each store that holds information for that object. One of the main features of the

[1]Message passing is the general term; in the system model, events (which include message events) are passed.

system model is its compositionality. This means that an object state can be described on an individual basis as well as in any (meaningful) group.

Transition Systems

We provide two different kinds of transition systems to define object behavior. Event-based state transition systems (see Sect. 4.7) are suitable to explain object behavior on a fine-grained level. Objects react to incoming events and their next computational activity is explicitly triggered by a scheduling event. The scheduling may be defined for groups of objects (belonging to the same processor, virtual processor, scheduling domain, etc.). Specific scheduling strategies, however, have not been defined yet.

As with object states, object behaviour can be described on an individual basis as well as in groups. Behaviour for compositions of groups of objects into larger components can be defined. For this purpose we use the time-aware version of state-transition systems, called timed STS (TSTS), see Sect. 4.8. Although asynchronous communication is assumed in the system model, the time-based approach allows the use of a simple abstraction on the time scale to look at communication as being synchronous. Communication between objects is dealt with by channels. Channels, on the one hand, help to compose groups of objects into larger units and hide their internal communication. On the other hand, UML provides linguistic constructs like "pins" in some of its diagrams; these pins resemble communication lines between objects and can be mapped to channels.

Further Extensions

Of course this system model that can be seen as a hierarchy of algebras may and probably should be extended by adding further functional machinery to ease description of the mapping of UML constructs to the system model. However, we wanted to keep the system model rather simple and therefore did not concentrate on this additional machinery very much. "Users" of the system model are really invited to add whatever they feel appropriate.

There are also a number of loopholes and particular variation points that can be further investigated by providing additional machinery to clarify a mapping of UML concepts to the system model.

3.6 USAGE SCENARIOS

After discussing the general approach to define the semantics of UML and highlighting the main characteristics of the system model, this section presents usage scenarios of a system model-based UML semantics.

Analysis

Assuming we have defined the semantics for UML using the system model, we are able to precisely express if a model A is well-formed, i.e., $sem(A) \neq \varnothing$. Similarly,

models A and B are consistent if $sem(A) \cap sem(B) \neq \varnothing$. It is well-known that for models to be well-formed a necessary but not sufficient condition is that they correspond to the language's grammar or metamodel. However, additional syntactic conditions, so called context conditions, need to be fulfilled. So, the challenge is to develop a set of context conditions *coco* that - if fulfilled - guarantees well-formedness, e.g., $coco(A,B) \Rightarrow Sem(A) \cap Sem(B) \neq \varnothing$. Analysing the well-formedness of models can then be reduced to checking syntactic conditions again. Unfortunately, also undecidable conditions exists that cannot be checked automatically but verification is needed.

Verification

As pointed out above, system model-based verification of UML models can be necessary to verify context conditions that cannot be checked automatically. In general, we are also interested in proving properties of concrete models using the UML semantics. The semantics characterizes all properties of systems s realizing model(s) A from which we then try to infer the property of interest ϕ, i.e., $\forall\, s \in Sem(A) \,:\, \phi\, s$. Verification can also be applied to prove transformations or generators correct. Assume, e.g., a transformation $\rightsquigarrow$ that refines model A to B, we have to show that $\forall A, B \,:\, A \rightsquigarrow B \Rightarrow Sem(B) \subseteq Sem(A)$.

Simulation

The system model deliberately is not defined in an executable way to support underspecification. It is, however, possible to resolve this underspecification and to encode the declarative specification into an executable simulator that is highly customizable with respect to semantic variation points [9]. Given a mapping from UML to the executable system model, we can validate models and experiment with different choices for semantic variation points via simulation.

Tool Support

The general approach to defining the semantics of UML has been outlined in Sects 3.2 and 3.3. To summarize, a precise and adequate semantics is made up of equally precise and adequate definitions for the syntax, the semantic domain, and the semantic mapping.

The most flexible way of defining these constituents surely is using pencil and paper. In order to keep most of the flexibility but to benefit from the advantages of a machine-readable semantics which can be (type) checked, used for automated verification etc., we use Isabelle/HOL [32] to formalize the system model definitions and also the semantic mapping. As a front-end for defining the syntax of the language, MontiCore [25], a framework for the development of modeling languages, is used. The overall approach is depicted in Fig 3.3.

1. The syntax is specified as a grammar in the MontiCore grammar definition language which basically is a context-free grammar.

2. The framework then generates a data type in Isabelle/HOL that represents the abstract syntax.

3. The semantics developer then uses this abstract syntax and the available formalization of the system model theories in Isabelle to encode the semantics of the syntactic constructs as predicates over system models.

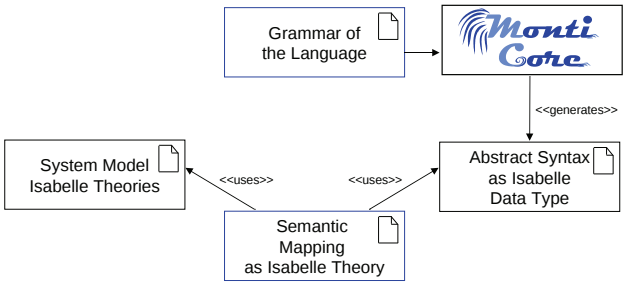

Figure 3.3 Approach with tool support

3.7 CONCLUDING REMARKS

In this Chapter, we described our approach to UML semantics and discussed the rationale underlying the system model definition. The system model describes structure and behaviour of object systems on a very detailed and fine-grained basis. It uses the general notion of (timed) state transition systems which is integrated with the data, control and event stores. As a general result of the system model theory, we have a complete description of how systems are decomposed into objects, what states objects may have and how objects interact. As motivated in the introduction, the mathematical theory is developed in layers, each building up an algebra that introduces some universe of elements, functions and laws for these functions. The detailed definitions can be found in Chapter 4.

The key features are support for underspecification, and a modular and flexibly extensible definition that is not biased by the choice of a concrete formal language or tool. Even the use of mathematical theories probably will bias the semantics a little but we hope as little as possible. Such bias easily creeps in and we carefully tried to avoid it. In particular, we do not address executability because this includes one of the biggest biases a modeling language can have: A model shall have the ability for underspecification. It shall be open for a specification of many different implementations. An executable semantics for an underspecified UML model must therefore necessarily contain implicit choices added by the semantic mapping.

To prevent the executability bias, we chose a specific style of description. This form of description allows us to leave quite a number of definitions open. We usually introduce a universe and then characterize the properties of its elements without fully

determining how many elements it has or how these elements look like. Sometimes, we only describe a subset of the elements and allow other kinds of elements to be in the universe as well.

This gives us the chance to specialize variation points according to specific situations. To put it in UML jargon, we could for example define a "system model profile" that specializes the general definitions to sequential, single threaded systems, to static systems without creation of new objects, or to systems without subclassing, etc. While the system model is an underlying basis for these kinds of systems, it does not provide such specialization directly; this is matter of further work. Indeed, as one of the results of this work, we have been able to make a number of variation points explicit. Although there are a lot more variation points to explore and their bandwidth to clarify, we regard this approach as a first important step to the formalization and clarification of variation points. On the other hand, the complexity of the system model shows that the integration of objects, threads, state-based behaviour and concurrency is complex, has many variations and is therefore somewhat arbitrary. It is particularly complex to model the possible interactions between these, leading us to the assumption that it is particularly difficult to master these not so well integrated concepts.

We also described usage scenarios and discussed how a system model-based UML semantics can be used to analyze, verify, and validate UML models.

The system model defined in [1] and the previous version [4, 5, 6] has actively been used to define the semantics of UML sublanguages like class diagrams [10] and Statecharts [11]. In [9] a simulator for UML models has been developed based on the system model definitions. This work has been carried out in the context of the DFG rUML project. In [13] UML actions are formalized using the system model as a semantic domain. The system model also forms the basis for characterizing the semantics of model composition [23] as part of the MODELPLEX project.

We wish to thank a number of colleagues, and especially Bran Selic, Michelle Crane, Jürgen Dingel, Gregor von Bochmann, Gregor Engels, Alain Faivre, Christophe Gaston, Sébastien Gérard, and Martin Schindler for their valuable help.

REFERENCES

1. Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*. Physica Verlag, Heidelberg, 1998.
2. Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a Formalization of the Unified Modeling Language. In *Proceedings of ECOOP'97 – Object Oriented Programming. 11th European Conference.* Springer-Verlag, LNCS 1241, 1997.
3. Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Modular Description of a Comprehensive Semantics Model for the UML (Version 2.0). Technical Report 2008-06, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

4. Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML – Towards a System Model for UML: The Structural Data Model. Technical Report TUM-I0612, Institut für Informatik, Technische Universität München, June 2006.

5. Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML – Towards a System Model for UML: The Control Model. Technical Report TUM-I0710, Institut für Informatik, Technische Universität München, February 2007.

6. Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML – Towards a System Model for UML: The State Machine Model. Technical Report TUM-I0711, Institut für Informatik, Technische Universität München, February 2007.

7. Jean-Michel Bruel and Robert B. France. Transforming UML models to Formal Specifications. In Pierre-Alain Muller and Jean Bézivin, editors, *International Conference on the Unified Modelling Language: Beyond the Notation (UML'98, Proceedings)*, volume 1618 of *Lecture Notes in Computer Science*. Springer, 1998.

8. Marco Cadoli, Diego Calvanese, Giuseppe De Giacomo, and Toni Mancini. Finite Model Reasoning on UML Class Diagrams Via Constraint Programming. In Roberto Basili and Maria Teresa Pazienza, editors, *AI*IA*, volume 4733 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007.

9. M. V. Cengarle, J. Dingel, H. Grönniger, and B. Rumpe. System-Model-Based Simulation of UML Models. In *Proceedings Nordic Workshop on Model Driven Engineering (NW-MODE 2007)*, 2007.

10. María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Technical Report 2008-04, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

11. María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Statecharts. Technical Report 2008-04, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

12. María Victoria Cengarle and Alexander Knapp. UML 2.0 Interactions: Semantics and Refinement. In Jan Jürjens, Eduardo B. Fernandez, Robert France, and Bernhard Rumpe, editors, *3rd Int. Wsh. Critical Systems Development with UML (CSDUML'04, Proceedings)*. Technical Report TUM-I0415, Institut für Informatik, Technische Universität München, 2004.

13. Michelle L. Crane and Juergen Dingel. Towards a Formal Account of a Foundational Subset for Executable UML Models. In *Models 2008*, 2008. to appear.

14. Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS Tutorials*, pages 70–98, 2003.

15. Gregor Engels, Christian Soltenborn, and Heike Wehrheim. Analysis of UML Activities Using Dynamic Meta Modeling. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2007.

16. Andy Evans, Kevin Lano, Robert France, and Bernhard Rumpe. Meta-Modeling Semantics of UML. In *Proceedings of Behavioral Specifications of Businesses and Systems*. Kluver Academic Publisher, 1999.

17. Harald Fecher, Marcel Kyas, Willem P. de Roever, and Frank S. de Boer. Compositional Operational Semantics of a UML-Kernel-Model Language. *Electr. Notes Theor. Comput. Sci.*, 156(1):79–96, 2006.

18. Ingo Feinerer and Gernot Salzer. Consistency and Minimality of UML Class Specifications with Multiplicities and Uniqueness Constraints. In *TASE*, pages 411–420. IEEE Computer Society, 2007.

19. Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. Developing the UML as a Formal Modelling Notation. In *Proceedings of the Unified Modeling Language. UML'98 Beyond the Notation. Mulhouse. Proceedings.*, pages 336–348, LNCS 1618. Springer, 1998.

20. Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, Technische Univerität München, 1996.

21. David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

22. Oeystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stoelen. STAIRS towards formal design with sequence diagrams. *Software and System Modeling (SoSym)*, 4(4):355–357, 2005.

23. Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, number 4530 in LNCS, pages 99–113, Haifa, Israel, June 2007. Springer.

24. C. Klein, B. Rumpe, and M. Broy. A stream-based mathematical model for distributed information processing systems the SysLab system model -. In E. Naijm and J.-B. Stefani, editors, *FMOODS'96, Formal Methods for Open Object-based Distributed Systems*. Chapman & Hall, 1996.

25. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Tools Europe*, 2008.

26. J. Küster, J. Koehler, J. Novatnack, and K. Ryndina. A Classification of UML2 Activity Diagrams. Technical report, IBM ZRL Technical Report 3673, 2006.

27. Sabine Kuske, Martin Gogolla, Ralf Kollmann, and Hans-Jörg Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 11–28, London, UK, 2002. Springer-Verlag.

28. Kevin Lano. A compositional semantics of UML-RSDS. *Software and System Modeling (SoSyM)*, to appear. DOI: 10.1007/s10270-007-0064-x.

29. Xiaoshan Li. A Characterization of UML Diagrams and their Consistency. In *ICECCS*, pages 67–76. IEEE Computer Society, 2006.

30. Azzam Maraee and Mira Balaban. Efficient Reasoning About Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2007.

31. Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

32. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002.

33. Object Management Group. Unified Modeling Language: Infrastructure Version 2.1.2 (07-11-05), 2007. http://www.omg.org/docs/formal/07-11-04.pdf.

34. Object Management Group. Unified Modeling Language: Superstructure Version 2.1.2 (07-11-02), 2007. http://www.omg.org/docs/formal/07-11-02.pdf.

35. Greg O'Keefe. Dynamic Logic Semantics for UML Consistency. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2006.

36. B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996. PhD thesis, Technische Universität München.

37. Stefan Sarstedt and Walter Guttmann. An ASM Semantics of Token Flow in UML 2 Activity Diagrams. In Irina Virbitskaite and Andrei Voronkov, editors, *Ershov Memorial Conference*, volume 4378 of *Lecture Notes in Computer Science*, pages 349–362. Springer, 2006.

38. Ken Satoh, Ken Kaneiwa, and Takeaki Uno. Contradiction Finding and Minimal Recovery for UML Class Diagrams. In *ASE*, pages 277–280. IEEE Computer Society, 2006.

39. Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

40. Thomas Stauner, Bernhard Rumpe, and Peter Scholz. Hybrid System Model. Technical Report TUM-I9903, Technische Univerität München, 1999.

41. Harald Störrle and Jan Hendrik Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke, editors, *Software Engineering*, volume 64 of *LNI*, pages 117–128. GI, 2005.

42. Ali Taleghani and Joanne M. Atlee. Semantic Variations Among UML StateMachines. In *MoDELS*, pages 245–259, 2006.

43. Michael von der Beeck. A structured operational semantics for uml-statecharts. *Software and System Modeling*, 1(2):130–141, 2002.

44. Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computer Science Series. MIT Press, Cambridge, Mass., 1993.

45. Xiangpeng Zhao, Quan Long, and Zongyan Qiu. Model Checking Dynamic UML Consistency. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 440–459. Springer, 2006.