

Chapter 2

Preliminaries

In this chapter, we provide the required background to keep this book self-contained. First, the concept of model-driven development with its realization in the form of the modeling language UML is introduced. Afterward, we discuss the applied methodologies, SMT solvers, and model finding.

2.1 Modeling Languages

In *model-driven development* (MDD, [1, 2]), the core of the development process is a *system model*. It serves as both specification and documentation and can be used as a basis for the implementation of the system. Originally, MDD was employed for software engineering only, but recently, several approaches on using modeling languages for hardware design have been proposed [3–5].

The field of modeling languages is vast depending on the domain where the language is to be employed [3, 6–9]. While often a graphical notation is preferred, purely textual languages exist as well. Some languages can be used independent of the system’s domain, whereas others are highly specialized.

The probably best-known and most popular modeling language is the *Unified Modeling Language* (UML, [10]) which was developed by the Object Management Group. The UML is well established in the field of software engineering and provides a large variety of different kinds of diagrams to form a system model. It is a mostly graphical language with the purpose of being easy to read and to understand for various stakeholders. While its original aim was the object-oriented design documentation, it has been developed to fit many different purposes.

In total, there are 14 different kinds of diagrams today. Each diagram focuses on a different aspect or viewpoint of the system. Basically, UML diagrams can be distinguished into two categories: *structural* diagrams and *behavioral* diagrams. Often, a combination of several diagrams from both categories is required to provide all the information on the system at hand.

In the following, we give an introduction to the parts of the UML used in this book. Furthermore, the Object Constraint Language (OCL) is presented which can be used to annotate UML diagrams.

2.1.1 The Unified Modeling Language UML

The UML offers various diagrams to describe the different aspects of a system. These diagrams combined are called a *model*. Basically, they can be divided into structural and behavioral description means. Here, we introduce only the diagrams used in this book: class diagrams, object diagrams, and activity diagrams.

In the following, we assume a bounded view on models as this is required in order to make formal methods applicable. For datatypes such as reals or integers which are defined as infinite in the UML, upper, and lower bounds for numbers are assumed. Additionally, when considering behavior, we only look at bounded sequences of operations and do not allow an infinite number of operation calls. These restrictions are justified by the fact that the model is to be implemented once the development process is completed and for an implementation, all of these boundaries are applied anyway.

2.1.1.1 Class Diagram

Class diagrams are one of the several ways to specify the structure of a system. Classes are used to model components which can be connected by associations. Attributes and operations are employed for a closer description of these classes.

Definition 2.1 A *class diagram* is a tuple $m = (C, R)$ composed of a set of classes C and a set of associations R . A *class* $c = (A, O, I) \in C$ of a class diagram m is a tuple composed of attributes A , operations O , and invariants I .

The set of all attributes of a class diagram m is given as $Attr^m = \bigcup_{c \in C} A$, the set of all operations as $Op^m = \bigcup_{c \in C} O$, and the set of invariants as $I^m = \bigcup_{c \in C} I$.

An n -ary *association* $r \in R$ of a class diagram m is a tuple $r = (r_{\text{ends}}, r_{\text{mult}})$ with *association ends* $r_{\text{ends}} \in C^n$ for a given set of classes C and *multiplicities* $r_{\text{mult}} \in (\mathbb{N}_0 \times \mathbb{N})^n$ that are defined as a range with a *lower bound* and an *upper bound*. An association end is annotated with a *role* e for the respective class in order to address the respective class at that association end, e.g., in a constraint.

Example 2.1 Figure 2.1a shows a class diagram composed of the classes Phone and CallApp. The two classes are connect via an association which indicates that each object of class Phone has to be linked to one object of class CallApp and vice versa. In the context of this association, the class Phone has the role phone and the class CallApp has the role callapp.

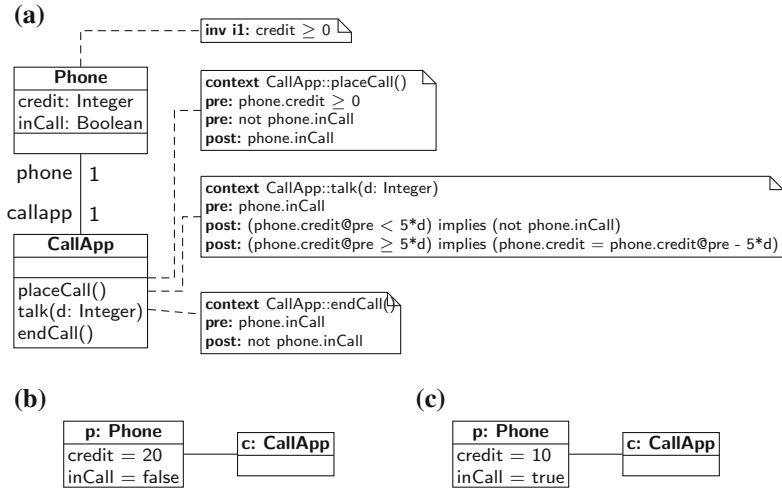


Fig. 2.1 Example of a class diagram and two of its instantiations. **a** Class diagram. **b** State σ_0 . **c** State σ_1

The class `Phone` itself is composed of the attributes $A = \{\text{credit}, \text{inCall}\}$, and the invariant $I = \{i1\}$. `CallApp` and, on the other hand, is composed of the operations $O = \{\text{placeCall}, \text{talk}, \text{endCall}\}$.

2.1.1.2 Object Diagram

Each class diagram describes a vast set of potential realizations or instances which are called *object diagrams*. An object diagram specifies a *system state* of the system modeled in the class diagram, whereas class diagrams provide the general structure and datatypes of a system, and object diagrams contain concrete objects in which the attributes are assigned precise values. Associations are instantiated as *links* between objects.

Definition 2.2 *Object diagrams* represent concrete system states in a model. A *system state* is denoted by σ and is composed of *objects*, i.e., instantiations of classes. The set of objects of a class `c` in a system state σ is given by the function `oid(c)`.

Example 2.2 Figure 2.1b shows a potential instance of the model from Fig. 2.1a. Each class has been instantiated once, resulting in the objects `p` of type `Phone` and `c` of type `callApp`. Both objects are connected via a link corresponding to the multiplicities defined in the class diagram, i.e., one object of each class is connected with an object of the other class.

2.1.1.3 Activity Diagram

To describe the control and data flow of a single operation or the system on the whole, *activity diagrams* can be used. They are a typical description mean to specify a system's behavior. In contrast to class diagrams, where an operation is simply specified in terms of name, parameter, and return value, activity diagrams provide much more detail. On a sufficiently low level of abstraction, they can be employed as basis for an implementation.

Each diagram represents an activity consisting of *action nodes*, *control nodes*, and *nodes for the object flow*. The control flow is modeled via *tokens* which are passed from node to node along the respective *edges*. Edges for control flow are depicted as arrows and can be annotated with *guard conditions*. Such a guard ensures that the respective edge can only be taken if the guard condition is true.

In the following, we consider the basic, most important modeling elements for activity diagrams used in this book. A complete description of activity diagram elements is given in the UML specification of the OMG [10].

- *Initial and final nodes*: The initial node contains a token which is handed to all outgoing edges. Final nodes destroy the tokens arriving there and either finish a single flow or the whole activity.



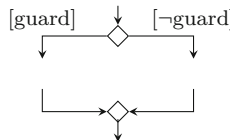
- *Action nodes*: Action nodes can contain various actions, such as assigning new values to variables, operation calls, or calling other activities.



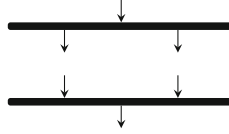
- *Send signal and accept event nodes*: They are used to model asynchronous behavior in such a way that by sending a signal, the processes with the corresponding accept event nodes are triggered.



- *Decision and merge nodes*: Each outgoing edge of a decision node is usually annotated with a guard condition so that the path can only be taken upon satisfaction of said condition. To join the paths started at a decision node, merge nodes can be applied. A decision node may occur without a corresponding merge node.



- *Fork and join nodes*: A fork node starts several flows to run in parallel while a join node unifies them again. Usually, join nodes are interpreted with an AND semantic so that all parallel flows have to be completed in order to continue. Other specifications such as OR joins, however, are possible and can be annotated with *{joinspec = ...}*. Fork and join nodes always have to be used pairwise.



These nodes concern only the control flow which is our main focus in this work. Activity diagrams offer several other concepts in order to model the data flow as well, such as *object nodes*, *parameters*, and *pins*. They are employed to pass objects or data from one node to another.

Example 2.3 Figure 2.2 shows the activity diagram for the operation `Phone::talk` (`d`: Integer). In order to take the edge from the initial node to the decision node, the attribute `inCall` has to be set to true. In case that `inCall` is false, the activity is stuck and cannot be executed.

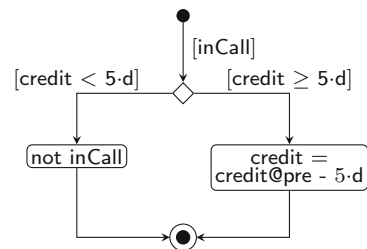
The decision node has two outgoing edges: If `credit` is less than $5 \cdot d$, then the left path is taken, leading to an action which sets `inCall` to false. Otherwise, the right path is taken and `credit` is decreased by $5 \cdot d$. Both paths lead to the final node.

2.1.2 The Object Constraint Language OCL

Class diagrams may be annotated in order to specify different kinds of constraints: global constraints and constraints over single transitions. Global constraints are called *invariants* and concern attributes or association ends. They have to be satisfied by each instantiation of the model, i.e., each created system state. Constraints over single transitions, on the other hand, can be used to specify the behavior of operations, following the *Design by Contract* paradigm [11, 12]. They are provided in the form of *pre-* and *post-conditions*. Preconditions have to be satisfied in order to be able to call the operation. After the execution of the operation, its respective post-conditions have to hold in the subsequent state.

Definition 2.3 *OCL expressions* Φ are textual constraints over a set of *variables* $V \supseteq A \cup R$. V is composed of the attributes A of the respective classes and the association ends contained in the relations R , but also further (auxiliary) variables (e.g., parameters of operations).

Fig. 2.2 Example of an activity diagram



An OCL condition $\varphi \in \Phi$ is defined as a function $\varphi : V \rightarrow \mathbb{B}$. They can be applied to specify the invariants of a class as well as the pre- and post-conditions of an operation. The domain $\text{dom}(\varphi)$ returns all variables of an OCL expression.

An *operation* $o \in \mathcal{O}$ is defined as a tuple $o = (\triangleleft, \triangleright)$ with preconditions $\triangleleft \in \Phi$ and post-conditions $\triangleright \in \Phi$, respectively. A precondition is a unary predicate, whereas a post-condition is a binary predicate depending on both the current and the previous states (denoted in OCL by adding @pre to the variables of the previous state). The sets of all pre- and post-conditions in a model m are given by Pre^m and Post^m , respectively.

The valid initial assignments of a class are described by a predicate $\text{init} \in \Phi$ which constitutes a set of valid initial states.

Example 2.4 In the class diagram in Fig. 2.1a, the invariant $i1$ states that `credit` always has to be greater than or equal to 0.¹ This global constraint has to be satisfied by all instantiations of the model. Assigning, for example, the value -5 to `credit` would not lead to a valid instance.

As for constraints over transitions, the operation `talk`, for example, is annotated with both pre- and post-conditions. The precondition requires that `inCall` in the associated `Phone` object is set to true; otherwise, it is not possible to invoke this operation. The post-conditions ensure that the `credit` remains greater than or equal to 0 in the subsequent state. One call costs 5 credit units, so if the duration of the call d times 5 is greater than the `credit`, the call is aborted by setting `inCall` to false. Otherwise, `credit` is reduced by the costs of the call.

In order to evaluate a model, it is crucial to particularly consider whether system states are valid or sequences of system states represent valid behavior. Such sequences of states are generated through the application of an *operation sequence*, denoted by $o_1 \cdot o_2 \cdot \dots \cdot o_k \in \mathcal{O}^+$, to the initial state. In short, an operation call is only valid if the operation is called in a state satisfying the operation's precondition and if the call results in a state satisfying the post-condition.

Definition 2.4 For a system state σ and an OCL expression φ , the evaluation of φ in σ is denoted by $\varphi(\sigma)$. A system state σ for a model $m = (C, R)$ is called *valid* iff it satisfies all invariants of m , i.e., iff $\bigwedge_{c \in C} I_c(\sigma)$. An operation call ω is valid iff it transforms a system state σ_t satisfying the precondition to a succeeding system state σ_{t+1} satisfying the post-condition, i.e., iff $\triangleleft_\omega(\sigma_t)$ and $\triangleright_\omega(\sigma_t, \sigma_{t+1})$. Such a state transition is denoted by $\sigma_t \xrightarrow{o} \sigma_{t+1}$, with o being the executed operation. A sequence of system states is called *valid* if all operation calls are valid.

Example 2.5 Figure 2.1b, c shows two valid system states in terms of object diagrams of the model from Fig. 2.1a. The states σ_0 and σ_1 cannot be connected via a single operation call: Since `inCall` is set to false in σ_0 , both `talk` and `endCall` are not applicable. The precondition of `placeCall` is satisfied, but σ_1 does not satisfy its post-condition.

¹For the sake of brevity, we will simply write `attribute` instead of `self.attribute` in the OCL constraints in this work when referring to an attribute of a class.

However, the state σ_1 can be reached from the state σ_0 by executing the following operation sequence: $p::\text{placeCall}() \cdot p::\text{talk}(2)$. The first state satisfies the precondition of `placeCall` and it is possible to create an intermediate state which satisfies the post-conditions. In this intermediate state, `credit` remains unchanged while `inCall` is set to true. From here, `talk(2)` is executable as the precondition is now satisfied and σ_1 satisfies the operation's post-conditions.

By using OCL constraints in class diagrams as shown above, it becomes possible to describe *both* structure and behavior in class diagrams. With the help of pre- and post-conditions, a *transition relation* can be derived from the class diagram which can be applied to the system's states. Still, actual behavioral diagrams are necessary as they provide much more detail on each operation and on the interaction between the system's components. The transition relation based on OCL constraints is much too general and contains no information on the data and object flow to be used as basis for an implementation.

2.2 Methodology: Model Finding

The employed methodology to solve the problems presented in this work is *model finding*. In this section, we present the two relevant notions: *structural* model finding and *behavioral* model finding. They are applied depending on the problem at hand and can be modified or varied by adding constraints.

2.2.1 Model Finding in General

The model finding problem in general deals with the question whether there exists an instantiation of a model satisfying a particular property. In the context of UML/OCL, we deal with class diagrams as models. A class diagram describes a vast state space of potential instantiations in terms of object diagrams. Without any further constraints, any object diagram is a valid instance of the class diagram. Global constraints on the model, i.e., OCL invariants, may restrict the search space since not all possible object diagrams satisfy all invariants of a model.

To obtain such an instantiation, various approaches have been proposed in the past [13–23]. A naive solution would be to enumerate all object diagrams, evaluate the given constraints on them, and pick one that satisfies all constraints. With regard to the size of the search space, however, such an approach would not be feasible. While it may be possible to list and evaluate all potential object diagrams for smaller models, with an increase in the number of classes and attributes, an automated method is required. In this work, we apply an SMT-based model finder as proposed in [24]. *Satisfiability modulo theories* or SMT is an extension to the *Boolean satisfiability problem* (SAT).

Definition 2.5 Given a Boolean formula $f : \mathbb{B}^n \mapsto \mathbb{B}$ in propositional logic, the SAT problem is to find a satisfying assignment to the inputs of f . If there exists an input assignment so that f evaluates to 1, then the function is *satisfiable*. If no such assignment exists, the unsatisfiability of f has been proven.

SAT was the first problem for which NP-completeness could be proven [25]. Even though it is a rather complex problem, various efficient solving engines exist for both SAT and its extension SMT [26–32]. The SMT problem extends the SAT problem in such a way that more complex formulas can be solved. Such an extension could be the use of bit-vectors instead of Boolean variables and formulas including inequalities or uninterpreted function symbols. This allows for a description on a higher level of abstraction, making the solving mechanisms applicable to more general and more complex problems.

As for the use in the context of model finding, Fig. 2.3 shows the work flow of an SMT-based model finder. Given is a model in terms of a class diagram, constraints in terms of OCL expressions, and a verification task. As the verification task can be rather specific, we will elaborate on this in the following sections. For now, let us assume that we can express this task in terms of a Boolean formula.

To solve the verification task, a *symbolic representation* of the model is generated as described in [24, 33]. This symbolic representation describes the complete search space in a compact way, i.e., all potential object diagrams which may be generated based on the class diagram. In the following, we will describe in more detail how to obtain such a representation.

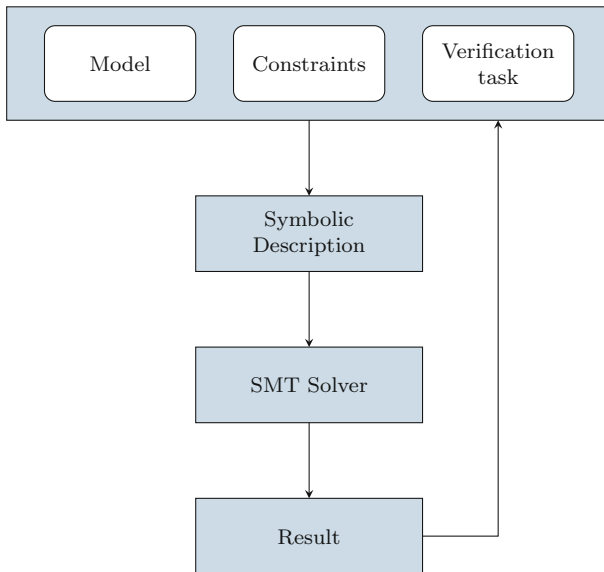


Fig. 2.3 Workflow of an SMT-based model finder

Fig. 2.4 Symbolic representation of a system state of the model in Fig. 2.1a

phone: Phone	calls: CallApp
$\alpha_{credit}^{phone} \in \mathbb{B}^8$	$\lambda_{calls}^{phone} \in \mathbb{B}$
$\alpha_{inCall}^{phone} \in \mathbb{B}$	
$\lambda_{calls}^{phone} \in \mathbb{B}$	

- In order to represent a system state, i.e., an object diagram, bit-vector variables for the classes' attributes and the associations between classes are introduced.
- For each attribute a of an object $o \in oid(c)$ of the class c , there is a bit-vector $\alpha_a^o \in \mathbb{B}^k$ with $k = \lceil \text{Id}(n) \rceil$. The value of n , i.e., the width of the bit-vector, depends on the datatype of a . For Boolean attributes, for example, n is equal to 2.
- Let $r = (r_{ends}, r_{mult})$ be an association between two classes c_1 and c_2 with e_1 and e_2 being the roles of c_1 and c_2 , respectively. Then, there is a bit-vector $\lambda_{e_2}^{o_1} \in \mathbb{B}^k$ with $k = |oid(c_2)|$ for each object $o_1 \in oid(c_1)$. This bit-vector describes the link between objects o_1 and o_2 of classes c_1 and c_2 . If the i th bit in this vector is set to 1, then object o_1 is linked to the i th object $o_2 \in oid(c_2)$. A bit set to 0 states that the respective objects are not connected.

Example 2.6 Figure 2.4 shows the symbolic representation of the class diagram from Fig. 2.1a. Here, one object for each class is assumed. Consider, for example, the object phone. For each attribute of the class, an α -variable has been generated. The variable α_{credit}^{phone} represents the integer attribute credit. As our model is bounded and we allow 8 bit integers at most, the length of this bit-vector is set to 8 accordingly. The variable representing the attribute inCall, α_{inCall}^{phone} , has a length of one as inCall can only be assigned two values.

In order to describe the link between the two objects, the variables λ_{calls}^{phone} and λ_{phone}^{calls} have been generated. The association between Phone and CallApp is a 1-to-1 relation; thus, there is only one option for each λ -variable: If it is set to 1, then the two objects are linked. Otherwise, they are not linked. In the case of a larger number of objects of either class, the length of the λ -vectors would have to be increased.

This symbolic representation can be handed to an SMT solver as an *SMT instance*. In order to answer the original problem—is it possible to instantiate the given class diagram—we solve the SMT instance with the help of a solver and obtain one of two possible results.

- SAT: The solver has found a valid assignment; i.e., it is possible to instantiate the model while satisfying all additional constraints.
- UNSAT: The solver could not find a valid assignment; i.e., it is not possible to instantiate the model due to contradictions in the model.

In case that a satisfying assignment has been found, the result has to be interpreted and retranslated to the modeling level. Based on the assignments to the bit-vectors, an object diagram is generated with the attribute values and links as determined by the solver.

Fig. 2.5 Satisfying assignment to the SMT instance from Fig. 2.4

phone: Phone	
α_{credit}^{phone}	$:= 00001010_2$
α_{inCall}^{phone}	$:= 0_2$
λ_{calls}^{phone}	$:= 1_2$

calls: CallApp	
λ_{phone}^{calls}	$:= 1_2$

Example 2.7 In Fig. 2.5, a satisfying assignment to the SMT instance shown in Fig. 2.4 is given. In order to derive an object diagram from this assignment, we need to interpret the values of the bit-vector variables. The vector α_{credit}^{phone} has been set to 00001010_2 . This means that the attribute *credit* of the object *phone* is set to the value 10. Accordingly, *inCall* is set to false since the corresponding bit-vector has been assigned 0_2 . The two objects *phone* and *calls* are linked since both λ -variables are set to 1_2 .

In the following, we will discuss the two different notions of model finding, structural, and behavioral model finding, in more detail.

2.2.2 Structural Model Finding

The *structural model finding* problem considers the question whether there exists a particular single system state σ satisfying a certain property. Note that for all kinds of structural model finding problems, the operations of the model are not of interest yet. A typical problem of this category is that of *consistency*. A state is considered to be *valid* or *consistent* if all invariants hold for all objects in that state, i.e., $I^m(\sigma)$. The consistency problem deals with the question if such a state exists, i.e., can the model be instantiated in such a way that all invariants are satisfied.

In order to solve the structural model finding problem, we have to extend the symbolic representation of the model as described above by a representation of the invariants. OCL invariants argue over attributes or association ends so they can be described as constraints over the α - and the λ -variables. More precisely, for each object $o \in oid(c)$ of a class c in a system state σ , we enforce that $I_c(\sigma)$; i.e., all invariants of the underlying class have to hold. This ensures that the solving engine only considers valid system states.

To determine the model's consistency, no further constraints are required while when dealing with other problems, a more complex representation may be required. The obtained SMT instance is solved and, in case of the consistency problem, may produce the following results.

- **SAT:** A system state σ which satisfies all invariants for all objects exists. The respective state is returned as a witness
- **UNSAT:** There is no system state σ so that all invariants hold for all objects due to an inconsistency in the model. The cause for this may be two or more contradictory invariants, a self-contradictory invariant, or contradictory multiplicities. To resolve such an inconsistency, debugging methods may be applied in order to help the designer determine the inconsistency's reason [34].

Example 2.8 In the case of the class diagram in Fig. 2.1a, the question of consistency can easily be answered. Here, a consistent model has to satisfy only the invariant i1 as well as the multiplicity constraint of the association. Both Fig. 2.1b, c are the witnesses for the existence of a valid system state as they both satisfy these constraints.

Different implementations for the structural model finding problem have been proposed in the past [17, 33, 35, 36].

2.2.3 Behavioral Model Finding

Where structural model finding concerns only a single system state, *behavioral model finding* can be seen as an extension which also considers the class diagram's operations. In particular, the verification tasks take the operations themselves or the system states generated by the operation's application into account. Thus, the transition relation we can generate based on the operation's constraints has to be included in the symbolic representation.

For this purpose, we assume a sequence of system states of length k , where k has to be predefined. Additionally, we assume an initial state σ_0 of the system. In order to describe the transition from one state to another in the sequence of states, we introduce a bit-vector for each step in the state sequence: $\omega_i \in \mathbb{B}^{\lceil \text{id}(\text{O}) \rceil}$ with $i < k$.

Now, each operation o in Op^m is assigned a unique binary representation by the function $\text{enc}(o)$, i.e., a number from 0 to $|Op^m|$ with $\text{enc}(\text{id}) = 0$ (with id being the identity function). If the vector ω_i is set to j , then the operation o with $\text{enc}(o) = j$ is called as the i th operation in the sequence of system states, connecting the states σ_i and σ_{i+1} . To enforce the operation's pre- and post-conditions, the following constraint is added to the SMT instance:

$$\omega_i = \text{enc}(o) \Rightarrow \triangleleft_o(\sigma_i) \wedge \triangleright_o(\sigma_i, \sigma_{i+1}), \quad (2.1)$$

Furthermore, to ensure that only legal values can be assigned to a vector ω , we use a constraint $\omega \leq |Op^m|$.

Based on this formulation, we can provide a symbolic representation of the transition relation as follows.

$$\bigwedge_{i=0}^k \mathcal{I}(\sigma_i) \wedge \bigwedge_{i=0}^{k-1} (\triangleleft_{o_i}(\sigma_i) \wedge \triangleright_{o_i}(\sigma_i, \sigma_{i+1})) \quad (2.2)$$

For each state σ_i in the state sequence, the invariants of the class diagram have to hold. Further, for each transition from a state σ_i to a state σ_{i+1} , there has to be an operation o_i whose constraints are satisfied.

Given this transition relation, the SMT solver attempts to generate a valid assignment to the states in the sequence of states as well as to the ω variables, representing

the operations which connect the states. Based on the obtained assignment, a single concrete operation call sequence can be generated. If the solver returns UNSAT, no such operation call sequence of length k starting from the state σ_0 exists.

In the following, we discuss some basic verification tasks which are typical for UML class diagrams. *Reachability* can be considered as a basic verification task in the field of behavioral model finding as all other problems can be expressed as a reachability problem. The question at its core is whether a particular system state σ can be reached through k operation calls. The respective verification task τ can be formalized as follows.

$$\bigwedge_{i=0}^k \mathcal{I}(\sigma_i) \wedge \bigwedge_{i=0}^{k-1} (\triangleleft_{o_i}(\sigma_i) \wedge \triangleright_{o_i}(\sigma_i, \sigma_{i+1})) \wedge \bigvee_{i=0}^k \sigma_i = \sigma \quad (2.3)$$

Here, the transition relation has been simply extended by an additional constraint (printed in **bold**). As before, we enforce that all invariants have to hold in all system states and that the pre- and post-conditions of the operations which are called between the states have to be satisfied. In addition to that one of the system states has to be the state σ . When we add this constraint to the SMT instance and solve it, the solver may return SAT and a witness in case that the state can be reached or UNSAT in case that no operation sequence leading to that state exists. A witness is a sequence of states of length k starting at σ_0 and leading to or including σ .

Example 2.9 For the model in Fig. 2.1a, we want to determine whether it is possible to reach a system state σ with `credit` = 0. The reachability of this state is of interest since it represents the lower bound of the range of `credit` according to the invariant `i1`. As initial state, we use the system state shown in Fig. 2.1b.

The state σ is reachable if there exists a sequence of operation calls of maximal length k from σ_0 to σ . For k , we have chosen 10.

Having represented the model as well as the two states and the operation calls in between as an SMT instance, the solver returns a satisfying assignment, i.e., a witness for an operation sequence. Only two operation calls are required to reach the desired state: `p::placeCall` · `p::talk(4)`. Other operation sequences to reach this state would be possible as well, e.g., `p::placeCall()` · `p::talk(3)` · `p::endCall()` · `p::placeCall()` · `p::talk(3)`.

Another typical verification task is the *executability* of an operation. An operation o is executable if there exists a pair of states in the sequence of states which is connected by the call of said operation. In other words, executability can be determined by proving the reachability of a pair of states (σ_i, σ_{i+1}) with $\triangleleft_o(\sigma_i)$ and $\triangleright_o(\sigma_i, \sigma_{i+1})$. Again, the solver will return SAT and a witness, i.e., a particular sequence of operation calls containing o , when the operation is executable and UNSAT otherwise.

Example 2.10 Consider again the first operation call sequence generated in Example 2.9. This sequence contains only the operations `placeCall` and `talk`. The operation `endCall` has not been called yet; thus, we want to find another operation call sequence containing also the missing operation call.

By employing the reachability problem, we can reformulate this problem as determining a pair of states σ_i and σ_{i+1} with $\triangleleft_{\text{endCall}}(\sigma_i), \triangleright_{\text{endCall}}(\sigma_i, \sigma_{i+1})$, and $i < k$. Again, we use the system state in Fig. 2.1b as initial state and set k to 10. Having added these additional constraints to the symbolic representation of the model, we solve the SMT instance and obtain a satisfying assignment, i.e., `endCall` is in fact executable. Based on the assignment, we retrieve the following operation call sequence as a witness: `p::placeCall() · p::talk(2) · p::endCall()`.

A third interesting problem is proving that a model is free of *deadlock states*. A deadlock state is a state in which no operation is executable, either because no precondition holds in the current state or because there is no subsequent state satisfying an operation's post-condition and all invariants. If such a state is reachable, the solver will return both the deadlock state and an operation sequence leading to said state. UNSAT, on the other hand, implies that the model is in fact free of deadlocks.

Example 2.11 To determine whether the model is free of deadlocks, we employ the reachability problem and search for a state σ_d such that

- none of the preconditions holds in σ_d which is only possible if `p.inCall` is set to false (otherwise `talk` or `endCall` are executable) and `(p.credit)` is less than 0 (otherwise `placeCall` is still executable), or
- one of the preconditions holds but there is no subsequent consistent state satisfying the corresponding post-conditions.

Adding these criteria to the symbolic representation of the model, the initial state σ_0 from Fig. 2.1b, and a maximal sequence length $k = 10$, the solver returns UNSAT. In other words, there exists no operation sequence of at most length 10 leading to a deadlock state. Of course, this does not imply that the model in general is free of deadlocks. We can only ensure that no such state is reachable within certain bounds.

Implementations for behavioral model finders have been realized, e.g., in [6, 24].

References

1. Schmidt, D.C.: Model-driven engineering. *Computer* **39**, 25–31 (2006)
2. Pastor, O., Espana, S., Ignacio, J., Aquino, N.: Model-driven development. *Informatik Spektrum* **31**, 394–407 (2008)
3. Weilkiens, T.: *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann, Massachusetts (2008)
4. Martin, G., Müller, W.: *UML for SOC Design*. Springer, Heidelberg (2005)
5. Vanderperren, Y., Müller, W., Dehaene, W.: UML for electronic systems design: a comprehensive overview. *Design Autom. Embed. Syst.* **12**, 261–292 (2008). doi:[10.1007/s10617-008-9028-9](https://doi.org/10.1007/s10617-008-9028-9)
6. Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: DynAlloy: upgrading alloy with actions. In: *Software Engineering*, pp. 442–451 (2005)

7. Leavens, G.T., Cheon, Y.: Design by contract with JML. <http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf> (2006)
8. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2010)
9. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Upper Saddle River (1996)
10. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Object Technology Series, 2nd edn. Addison-Wesley Professional, Boston (2004)
11. Meyer, B., Nerson, J.M., Matsuo, M.: Eiffel: object-oriented design for software engineering. In: European Software Engineering Conference, pp. 221–229 (1987)
12. Meyer, B.: Applying ‘design by contract’. *Computer* **25**, 40–51 (1992). doi:[10.1109/2.161279](https://doi.org/10.1109/2.161279)
13. Bauer, J., Damm, W., Toben, T., Westphal, B.: Verification and synthesis of OCL constraints via topology analysis. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Applications of Graph Transformations with Industrial Relevance. Lecture Notes in Computer Science, vol. 5088, pp. 361–376. Springer, Heidelberg (2007)
14. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL operation contracts. In: Leuschel, M., Wehrheim, H. (eds.) Integrated Formal Methods. Lecture Notes in Computer Science, vol. 5423, pp. 40–55. Springer, Heidelberg (2009)
15. Cadoli, M., Calvanese, D., Giacomo, G.D., Mancini, T.: Finite model reasoning on UML class diagrams via constraint programming. In: AI*IA, pp. 36–47 (2007)
16. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, independence and consequences in UML and OCL models. In: Tests and Proofs, pp. 90–104 (2009)
17. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into USE. In: Objects, Models, Components, Patterns, pp. 290–306 (2011)
18. Li, X., Liu, Z., He, J.: Consistency checking of UML requirements. In: Engineering of Complex Computer Systems, pp. 411–420 (2005)
19. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to alloy. *Softw. Syst. Model.* **9**, 69–86 (2010)
20. Baruzzo, A., Comini, M.: Static verification of UML model consistency. In: Model Design and Validation, pp. 111–126 (2006)
21. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. *Artif. Intell.* **168**(1–2), 70–118 (2005)
22. Chiorean, D., Pasca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML models consistency using the OCL environment. *Electron. Notes Theor. Comput. Sci.* **102**, 99–110 (2004)
23. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. *Electron. Notes Theor. Comput. Sci.* **115**, 39–47 (2005)
24. Soeken, M., Wille, R., Drechsler, R.: Verifying dynamic aspects of UML models. In: Design, Automation and Test in Europe, pp. 1077–1082 (2011)
25. Cook, S.A.: The complexity of theorem-proving procedures. In: Symposium on Theory of Computing (1971)
26. Goldberg, E.I., Novikov, Y.: BerkMin: A fast and robust Sat-solver. In: Design, Automation and Test in Europe, pp. 142–149 (2002)
27. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conference, pp. 530–535 (2001)
28. Biere, A.: PicoSAT essentials. *J. Satisf., Boolean Model. Comput.* **4**, 75–97 (2008)
29. de Moura, L.M., Björner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for Construction and Analysis of Systems, pp. 337–340 (2008)
30. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT solver. In: Computer Aided Verification, pp. 299–303 (2008)
31. Wille, R., Fey, G., Große, D., Eggersgluß, S., Drechsler, R.: SWORD: A SAT-like prover using word level information. In: Very Large Scale Integration of System-on-Chip, pp. 88–93 (2007)
32. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Tools and Algorithms for Construction and Analysis of Systems, pp. 174–177. Springer, Heidelberg (2009)

33. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using boolean satisfiability. In: Design, Automation and Test in Europe, pp. 1341–1344 (2010)
34. Wille, R., Soeken, M., Drechsler, R.: Debugging of inconsistent UML/OCL models. In: Design, Automation and Test in Europe, pp. 1078–1083 (2012)
35. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Massachusetts (2006)
36. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Software Testing, Verification and Validation, pp. 73–80 (2008)

Automatic Methods for the Refinement of System
Models

From the Specification to the Implementation

Seiter, J.; Wille, R.; Drechsler, R.

2017, VIII, 94 p. 30 illus., 5 illus. in color., Softcover

ISBN: 978-3-319-41479-9