Exploiting Practical Limitations of UML Diagrams for Model Validation and Execution

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fachbereich Informatik
Fernuniversität in Hagen
Universitätsstraße 1, D-58097 Hagen
steimann@acm.org

Heribert Vollmer
Theoretische Informatik
Institut für Informationssysteme
Universität Hannover
Appelstraße 4, D-30167 Hannover
vollmer@thi.uni-hannover.de

Abstract. We suggest a framework for UML diagram validation and execution that takes advantage of some of the practical restrictions induced by diagrammatic representations (as compared to Turing equivalent programming languages) by exploiting possible gains in decidability. In particular, within our framework we can prove that an object interaction comes to an end, or that one action is always performed before another. Even more appealingly, we can compute efficiently whether two models are equivalent (aiding in the redesign or refactoring of a model), and what the differences between two models are. The framework employs a simple modelling object language (called MOL) for which we present formal syntax and semantics. A first generation of tools has been implemented that allows us to collect experience with our approach, guiding its further development.

Keywords: modelling, modelling language, validation of models, UML

1 Introduction

Two major strains of UML's further advancement are followed rather vigorously: making it precise, and making it executable. Quite obviously, the former is a prerequisite for the latter: a language lacking precise semantics cannot be executable or, put positively, any executable variant of UML must necessarily be precise. On the other hand, making UML precise does not automatically mean that the result is executable: much theoretical work has been invested in translating UML to some other formalism that, although coming with proper (mostly denotational) semantics, is equally non-executable (see [32] for a collection). This is in contrast to the needs of the UML user community which, although agreeing to the need for precision on theoretical grounds, is much more interested in an executable UML, allowing models to be validated directly against user requirements, substituting for the often expensive prototyping.

Rather than trying to turn UML into a full-fledged (graphical) programming language, we note that *modelling is not programming*; in particular, that diagrammatic models have some properties that are very different from general programs, and that these differences put UML (as well as other graphical modelling languages) in a different class than general purpose programming languages. To be able to exploit these particular properties, we define a special programming language, and provide a mapping from a subset of UML diagrams to this language. This mapping not only enables the execution of models, but also allows the proof of certain properties and their transfer back to the original models.

The remainder of this paper is organized as follows. After presenting the rationale of our work, we discuss what we have observed as the limitations of graphical modelling. Based on these observations we define the core of our executable language as the target of mapping certain UML diagrams, and specify its formal syntax and semantics. Next, we discuss the theoretical properties of this language, which justify its design. The translation of UML diagrams to programs is then demonstrated by presenting a set of sample diagrams and corresponding code. Some comments on the design process of the formal framework and the tools we created so far are to give an idea of where we are headed. A discussion and the placement of our work in a larger context conclude our contribution.

2 Rationale

Analogous to the validation of software, validation of object models means checking them against the customers' requirements. To paraphrase an old slogan of software engineering, model validation ensures that one is building the right model, as opposed to ensuring that one is building the model right (the latter being called verification) [16]. Unlike verification, validation invariably involves the customer and requires her or his understanding of what is being modelled. Even though UML diagram types have generally been designed to be intelligible, at least to the technically trained, the more revealing a model is to be, the more complex it becomes. In fact, interpreting a diagram is not necessarily simpler than reading a piece of code (and its creation is just as error-prone): the validation of a model is an intellectual act, and the ill-performance of it places any software project at substantial risk.

Validation of a model greatly benefits from executing it, from "seeing what it does". Validating a model through executing it benefits from the fact that the objects and links in models are limited to fairly small numbers, and that their interaction specifications are only exemplary in character: unlike a complete program, a model seems to be explorable in full. But even if the number of possible different "runs" of a model remains too large for a human validator, it will typically not be for the automatic proof of certain desired properties, particularly not if efficient algorithms are applicable.

While the type-level specifications of UML diagrams generally describe infinite structures, the instance level which is the basis for behavioural specifications, is finite in character. For instance, UML

interaction diagrams specify the collaboration of a finite set of (possibly unnamed, i.e., anonymous) objects. In fact, as will be argued in more detail below even multiobjects can be considered as a single object since all objects they represent share the same structure (which is why they can be represented by a single symbol in the diagram). Thus, the interpretation of any behavioural specification involves only a finite number of states.

Statecharts (complementing interaction specifications by the behaviour of individual objects) are finite by definition: firstly, they model the behaviour of a single (albeit anonymous) object, and secondly, because the number of states is per definition finite, they too require no infinite semantic domain. All in all, it appears that object-oriented modelling can do with a finite (and generally also rather limited) number of objects and states. This is in sharp contrast to object-oriented programming, in which the number of objects has no theoretical bound. Therefore models have stronger formal properties than general programs; therefore our conclusion that modelling is not programming.

In order to be able to exploit the observed properties of modelling, we have to select a semantic formalism which shares these properties. Because there is a strong natural relationship between certain diagram types of UML (in particular class, object, and interaction diagrams) and object-oriented programming, our work is greatly facilitated by mapping UML diagrams to programs in a special *Modelling Object Language*. By designing this language—called MOL hereafter—to be as primitive as possible, we avoid tying the interpretation of UML diagrams to language peculiarities (such as a specific type system, call bindings, etc.). This of course comes at the price of splitting the definition of the semantics of UML between the definition of the semantics of MOL and the definition of the required model compiler. On the other hand, it allows us to cleanly separate between the general properties of object-oriented modelling and the many-facedness of UML with its numerous possibilities for introducing extensions and alternative meanings; it allows the easy introduction of alternative mappings from diagrams to programs.

3 Practical limitations of drawings

Our work is based on a number of assumptions. First, we assume that modelling is not programming, i.e., that the purpose of a model is different from that of a program. In particular, we assume that models are not used to compute functions, be it numeric (such as factorials) or symbolic (such as text processing). Instead, we assume that functions, if needed, are specified outside the model and can be used as modelling primitives. The model itself specifies the possible interactions (communication) between the user and the objects of a system, and the changes in the object (or data) structure these interactions produce. We are well aware of the fact that any state transformation can be regarded as a function (from the current state and the triggering input to the next state); that indeed the state transi-

tion function is a function, but this function is special in that it is recursively applied to itself, with the end of the computation being controlled by user (or outside system) interaction.

The other key assumption we make is that there are practical constraints for drawings that do not equally apply to written specifications such as programs. Although we acknowledge that graphical and textual languages are equivalent in principle, we maintain that drawings are used to provide overview, especially through making the connections between different parts of a system visible (the inevitable lines in the ubiquitous lines and boxes diagrams), whereas textual specifications emphasize modularity, i.e., tend to form small clusters of specification (such as axioms, rules, or subroutines) which can freely be combined into larger ones. Indeed, the connectedness of drawings comes at the price of genericity: while a box can be a placeholder, i.e., stand for any object satisfying certain conditions, an attached line invariably connects the object to another (even if the objects are not named). Because connectedness is transitive, the lines in a diagram often impose more state than wanted, requiring alternative diagrams or special notations if variability is needed. Annotations like `{xor}` and `{new}` constraints or «`create`» and «`delete`» stereotypes may provide some relief, but their coordination (if more than one such constraint or stereotype occurs in the same diagram) is difficult and confusing. The connectedness of a diagram and resultant overspecification is best counteracted by reducing its size (the number of connected elements shown on a single diagram), but increasing the modularity of drawings to levels comparable to that of textual specifications makes them lose their overview character, counteracting their very purpose.

Resultant from the reduced genericity of drawings is indeed a reduced number of captured states which, under certain conditions, can even be confined to a finite (and usually also fairly small) number. We shall take a look at the necessary preconditions in some more detail.

3.1 Instance level, type level, and the world between

As suggested by the Meta-Object Facility (MOF) [30], the UML standard tries to make a clear distinction between specifications on the instance and on the type level. The class diagram for instance is a static structure specification on the type level, whereas the object diagram is one on the instance level. The distinction is less clear for interaction diagrams, which can be specified on the instance and on the specification level: the latter is not called type level presumably because the creators of UML felt that collaboration roles (as classifiers) were somewhere between the type and the instance level.

In general, specifications on the type level are generic since they involve whole sets of objects, whereas those on the instance level are concrete, involving only identifiable individual objects. The genericity of the type level is bound to the notion of type, which is mathematically defined as a predicate deciding whether a given object is an instance of that type or not. This includes relationships, which are interpreted as types of higher than unary arity [41]. Because the sole use of types leads to

combinatorial explosion and accordingly underspecified models, additional standard predicates (such as multiplicities, navigability, etc.) have been added to the syntactic repertoire of the type level, constraining the number of possible instantiations of a type-level specification. Mathematically, however, these constraints cannot be expressed through sets (the types) alone; they require variables ranging over the instances of a type. In fact, even the semantics of a very simple class diagram cannot be expressed without resorting to variables: does the fact that type `A` is related to type `B` via association `r` mean that all instances of `A` are related to all instances of `B` all the time? Certainly not, but expressing what it really means invariably involves variables.

On closer inspection, one discovers that instance level specifications are not free of variables, either: in an object diagram for example, objects may remain anonymous, in which case they must be considered placeholders for concrete objects. Therefore, we do not distinguish between type and instance level specifications, but—following mathematical logic—between *non-ground* and *ground specifications*, depending on whether the specifications do or do not contain variables.[1] In this vein, a class diagram is a non-ground specification whose variables are constrained by types (and, possibly, also other expressions), whereas an object diagram with named objects is ground. Interaction diagrams on the instance level are also ground (if all objects are named), but those on the specification level are characteristically non-ground: their classifier roles are variables whose values are constrained to possess the links and methods shown in the diagram (with type constraints being optional).

Non-ground specifications can be instantiated to yield ground specifications. Since in object-oriented terms, instantiation is an explicit act that is itself the subject of modelling, we require that for the instantiation of a non-ground diagram (instantiation here denoting the assignment of concrete objects to placeholders) all objects must pre-exist. For reasons given below, this includes those whose placeholders are annotated for explicit object creation. Since object structures are injected into a model through ground diagrams, the number of possible instantiations of a non-ground diagram is limited. Note that repeated object creation for the same placeholder would require this to be a multiobject, but again, since all objects comprised under the multiobject share the same structure (as expressed by the structural embedding of the multiobject in the diagram), they can be considered as one.

3.2 The open world assumption of modelling

The state of an object-oriented system is defined as the combined state of its objects, plus the program pointer(s) and call stack(s). The state of an object is determined by its attributes and the links it has to

[1] As a matter of fact, type information is nothing but a special constraint on variables; a result from first order logics shows that many-sorted logics is no more expressive than its uni-sorted ancestor, since a typed variable declaration $x{:}T$ can always be transformed to expressions involving the corresponding type predicate $T(x)$. The insight that the formal parameter and return types of methods are just special pre- and post-conditions is based on the same considerations.

other objects. Since we will interpret links as pointers and an object has no knowledge of its incoming pointers, the latter do not contribute to an object's state. To simplify matters, we will regard attributes as notational shortcuts that can always be translated to links to other objects (including immutable or value objects) if needed.

Given this definition of state, every ground diagram makes a set of statements about the involved objects' states plus, depending on the diagram type, what can happen to them once they are in the given states. It can be read as: "if the objects are in the given state, the following may happen". (In case of static diagrams the statement reduces to "the objects may be in the given states".) Every non-ground diagram that can be obtained from a ground one by replacing objects with placeholders (variables) transfers the same statement to sets of objects, which are constrained to being able to adopt the shown states. Type information may contribute to these constraints.

Now a model is an abstraction, meaning that it does not show everything. While this is particularly true for single diagrams (whose nature it is to show only specific views of the system being modelled), it usually also holds for the set of all diagrams provided. Therefore, that something is not drawn does not mean that it is not there (unless of course stated otherwise). For instance, a diagram may show two objects that are not linked, even though a link may in fact exist (albeit it is of no interest in the particular diagram). Therefore, diagrams must generally be assumed to show projections of the involved objects states, where each such projection may in fact stand for infinitely many different states, all sharing the same projection. We refer to the fact that a diagram only shows what is there, not what is not there as the *open world assumption of modelling*; it allows a finite state model to represent a real system with infinitely many states.

That the open world assumption is necessary for modelling derives from the fact that for a complete system specification, objects of different diagrams must be linked somehow. In order to be able to show excerpts in the form of single diagrams or views, some of the links must be broken, i.e., not shown. Typically, diagrams will contain central objects (all relevant links of which are shown) and peripheral objects (the links of which are shown on other diagrams).[2] This is important, since the unshown links of peripheral objects introduce variability even for ground diagrams. In particular, each peripheral object is a potential variation point (see below).

The fact that the open world assumption of modelling applies equally to ground and to non-ground diagrams has an important implication for the latter, i.e., diagrams containing placeholders. Since what is shown always pertains to all possible assignments, the actual objects taking the places of the placeholders may vary only in the state (i.e., attributes and links) not shown. However, since this state

[2] Note that the distinction between central and peripheral objects cannot be based on a single diagram, since the unshown links of peripheral objects must be considered as insignificant to the modelled aspect as those unshown of the central objects. Whether or not an object is peripheral is only determined by whether or not it is involved in the continuation of the diagram in others, as discussed in Section 3.3.

is not shown, the possible variation must be considered irrelevant for the modelled aspect. Therefore, non-ground diagrams differ from ground diagrams only in the respect that the actual objects (values of the placeholders) are not named—the structural constraints expressed (the "if" part of the statements) and the possible variability are the same. There is however an increase in the variation induced by the continuation of peripheral objects, as will be argued next.

3.3 Continuation, recursion, and infinity

In ground diagrams, all objects are named. A ground diagram can be continued in another if names of objects coincide (the names serving as so-called co-references). The state constraints of each diagram then overlay each other. If alternative diagrams with identical objects exist, continuation introduces variability even to ground specifications; a diagram can be continued by any one of the offered alternatives. Since the state specifications of the different diagrams combine to form a larger one, the modelled state space grows combinatorially with the number of alternative continuations. Continuation can even introduce circles: such is the case for example if one diagram is continued in a second which is continued in a third which in turn is continued in the first. However, continuation of ground diagrams cannot lead to an infinite number of states, because the set of such diagrams is always finite, as is the number of objects contained in them.

In ground diagrams peripheral objects are named and continuation of a diagram in others is uniquely determined by these names. For non-ground diagrams, however, and in the absence of co-references between variables continuation constraints are only implicit in the specified state spaces (including the types, if provided) of placeholders. In fact, the type of a placeholder is all that remains to join different diagrams if a peripheral object has no additional state specification, e.g. if it has incoming links only. Note that role names (or classifier roles as employed in collaboration diagrams) are of no help, since the roles of an object (or placeholder thereof) in different diagrams are typically different. Therefore, the number of diagrams that can be connected is usually much larger for non-ground than for ground diagrams; in fact, if a diagram can be connected to itself, it can even generate infinite structures, as the following considerations show.

Figure 1 a) shows a non-ground collaboration diagram with five variables. Note that the variables are distinguished only by their location, not by their labels; for any linearization (transformation into textual form), they must be given unique names. Also note that the diagram contains a somewhat unusual specification of a conditional (branching): generalization arrows indicate that the central component placeholder can be either a leaf or itself a composite. An `{xor}` constraint could have been used instead, but would not have exploited the subtyping of `Component`.

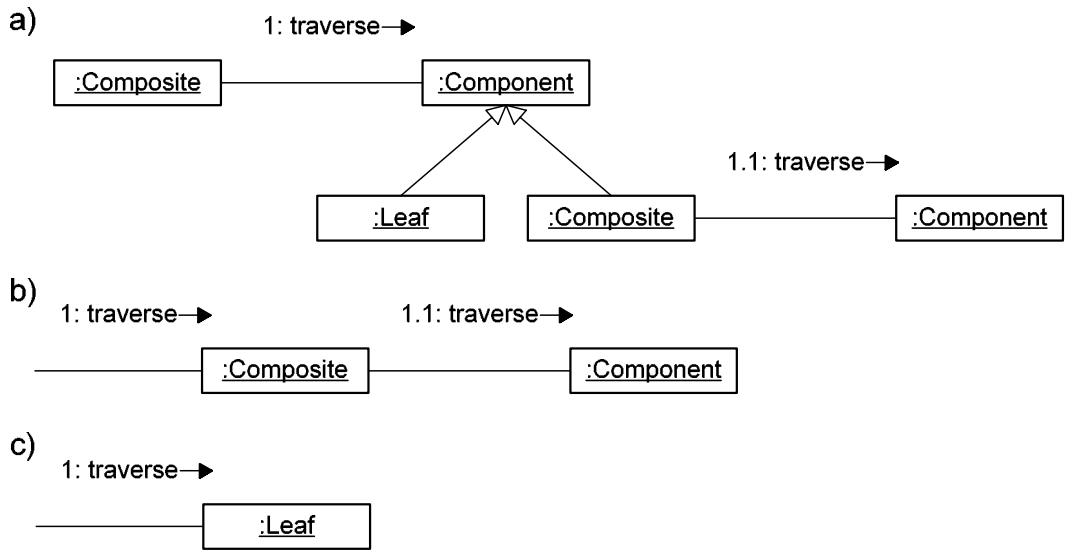


Figure 1. Non-ground specification of a recursive structure (COMPOSITE PATTERN) and its traversal as a collaboration diagram. a) Explicit recursion of one level. b) Implicit infinite recursion only. c) Its termination.

The diagram of Figure 1 a) describes a variant of the COMPOSITE PATTERN [17], which is a recursive structure, which is mirrored by the fact that it can continue itself. In fact, there are two possible self continuations: one regards the rightmost component and the leftmost composite placeholder as peripheral objects or continuation points, while the other regards each composite/component-pair (together with their links) as a continuation point. The former continuation most likely does not express what is meant, though, since it implies that only every second component can be a leaf; yet there is nothing in the diagram that excludes this possible continuation. In fact, as will be argued below recursive continuation of diagrams is often the source of modelling bugs and therefore to be treated with caution.

The explicit variation in Figure 1 a) can be eliminated by dividing it into two separate diagrams. In fact, Figure 1 a) can itself be regarded as the continuation of two smaller diagrams, namely Figure 1 b) and c). However, while Figure 1 a) shows an explicit one-level structural recursion by repeating (parts of) the same structure within the same diagram, the same can only implicitly be derived from Figure 1 b) and c), namely through searching for possible continuations in the pool of all diagrams. While the explicit structural recursion must be finite (because it must fit on one diagram), the implicit is typically not, leading to infinite structures.

Generally, we maintain that in modelling significant structure introduced by the continuation of a diagram is unintended and likely to be the source of modelling bugs. The continuation of UML metamodel diagrams performed in [39] and the flaws thus discovered may be considered representative of the problem. Considering that structural recursion introduced through diagram self continuation is—except perhaps for rare cases—not necessary to grasp the structure of a system (at least not for valida-

tion purposes), it should be flagged a potential modelling error rather than being accepted as a valid means of expression.[3]

3.4 Implications

Assuming that the open world assumption of modelling holds, i.e., that all unshown differences in state (that is, all unshown links) are insignificant to the modelled problem, we can draw the following conclusions:

1. A single ground diagram represents a finite number of states (precisely one state in absence of constraints or stereotypes introducing variability).
2. The set of states that can be generated by the continuation of ground diagrams is also finite. This follows from 1 and the fact that continuation of ground diagrams can introduce no structural recursion (only cyclic structures).
3. A single non-ground diagram can represent as many different (sub)states of a system as there are possible instantiations (variable assignments). The number of possible instantiations of a single diagram is limited by the number of objects in a model, which is finite. Recall that `{new}` constraints or `«create»` stereotypes create only one new object per instantiation, given that the instances of multiobjects all share the same state and can therefore be regarded as one.
4. Continuation of non-ground diagrams can lead to an infinite number of states if the same diagram is instantiated repeatedly with both same and new objects (such that the same objects in different instantiations mark continuation, whereas the introduction of new objects avoids cycles). However, considering that aspects of interest are usually modelled in one diagram each, repeating the same aspect infinitely through structural recursion does not add to what the model is to express. Since it bears the danger of introducing errors, it should be avoided.

It follows that except for rare cases, models specify only a finite number of possible states, covering the potential infinity of the modelled system through the fact that each modelled state is a projection of an unlimited number of real states. This has important implications for our further reasoning.

[3] In a way, modularity and self-continuation of the diagrams of a model correspond to paragraphs and their ending in “and so forth” in written specifications: they explain in sufficient explicitness the structure of interest and leave the possible continuation implicit, because nothing new is to be learnt from it. For instance, Figure 1 a) shows all significant aspects of the pattern in one diagram, and nothing of significance is learnt from a possible continuation. This is different for Figure 1 b) and c), which only jointly express the structure to be modelled.

4 UML-A and the definition of MOL

UML in its entirety is not a single language, but a whole family of languages, in particular one that is open to extensions. Coming to grips with UML semantics is therefore inherently difficult; in fact, one can even argue the meaning of UML is defined by its use, granting it the status of a natural (rather than a formal) language.

In a comment on the presentation of UML's next major release, UML 2.0, Dennis de Champeaux wrote:

»*The entirety of UML is not required for dealing with problem understanding. A subset of UML—call it UML-A (for UML OO Analysis)—should be used to "rewrite" the unformalized requirements document into a formal version, without committing to how the system would operate. UML-A should be simple enough that the sponsor helps validate that the model captures the intent expressed by the requirements; for example, the sponsor should be able to confirm that use cases formulated in plain English (and optionally captured in diagrams) are faithfully represented in interaction diagrams and scenario diagrams.*« [10]

We pick up de Champeaux's argument and try a first definition of such a language UML-A, one that keeps the major properties and most of the appearance of UML, and at the same time allows us to fully exploit what we have observed above. Currently, this UML-A covers use case diagrams, class diagrams, object diagrams, and interaction diagrams (with a focus on collaboration diagrams). Statecharts have been excluded so far, since the problem of relating their abstract states to the concrete states of an object (as reflected in the links the object has) is as yet unsolved ([39]; *cf.* Discussion).

Our current definition of UML-A is constrained by two fundamental conditions:

1. all diagrams of a model must be integrateable into a single system specification, and
2. the specification must be executable.

Deviating from the usual procedure of language design we therefore take an unusual approach and start with the definition of our chosen semantic formalism, MOL, for which we maintain that it is sufficiently expressive for software modelling, given the assumptions of Section 3. We then define UML-A as that subset of UML that easily maps to our semantic formalism, and furnish it with certain extensions supporting model integration.

There is a wealth of formalisms that can serve as the target of a semantic mapping for UML. For our purpose of model validation through execution, we are looking for one that is executable and whose execution model mirrors as much of UML's intended semantics as possible, so that the mapping becomes straightforward. In fact, stepwise execution of a model should be reflected in model animation, i.e., in the animated graphical display of methods exchanged and the structure altered, as suggested in [42]. Since behaviour specification in UML is heavily influenced by the semantics of object-oriented programming (with objects exchanging messages and changing their state in response), a

primitive (and free of peculiarities such as certain binding conventions) object-oriented programming language seems an obvious choice. We have designed our modelling object language MOL to be just that.

We are well aware of the fact that UML semantics is intentionally left open so that there are as many different semantics as there are different uses of the language, yet it must be clear that any executable formalism can only have one semantics—only which one is subject to debate. In this vein, the one we chose and present here must only be seen as a first suggestion—the discussion is still open and our work may evolve in a different direction as our experience grows. Yet, as will be detailed in Section 5, our current choice is not without advantages.

4.1 Language elements

Objects. In MOL all data elements are objects. Each object has a name by which it is represented in the program text and which identifies the object uniquely.[4] This is owed to the fact that in a model an object is represented by a unique graphical element; it is in contrast to general object-oriented programming, in which object names (pointers) are stored in variables, but do not themselves appear in the program text.

The creation of a new object is an explicit act, in MOL introduced by the keyword `new` followed by the name of the new object. Of repeated `new` statements with the same name, only the first is effective. This means that object creation repeated in loops (multiobjects) creates only one object. Created objects are destroyed through the keyword `delete` followed by the name of the object. Again, repeated deletion of the same object remains ineffective. Since objects can be the values of attributes and variables, deletion of an object can make these attributes and variables undefined.

Attributes. Even though suggested differently by the semantics of class diagrams, the data model of object-oriented modelling is inherently navigational and not relational. Therefore, in MOL all interrelations between objects are represented through attributes.

An attribute maps an object to another, called the *attribute value* of the object. The attribute value of an object is represented as `object1.attribute1[]`. New values can be assigned to an attribute by using the assignment operator, `:=`.

In object-oriented models, attributes can have a multiplicity greater than one (or, more prominently, an object can be linked to several others by way of the same association). This is usually interpreted

[4] It follows that the number of dynamically created objects in a program is always finite, since it cannot be larger than the number of names. This is in contrast to most object-oriented programming languages, which allow unnamed objects to be created and subsequently be referred to through variables. One might consider lifting this restriction for future versions of MOL, but first it is necessary to find an alternative mechanism for restricting the number of objects in a system.

as the attribute being set-valued. However, since the set holding the values is itself an object, this representation introduces a level of indirection not justified by the modelling language: whether an attribute or association end is of 0..1, 1, *, or of any other multiplicity is a matter of degree, not a fundamental difference. Therefore, we regard single-valued attributes as special cases of many-valued attributes and, generally, attributes as binary functions, where the first argument is the object being attributed (the owner of the attribute), and the second is a qualifier of the attribution. In MOL, `object1.attribute1[object2]` stands for `object1`'s value of `attribute1` qualified by `object2`. MOL's attributes therefore naturally account for associative arrays with normal (integer) arrays as their special cases.

Prior to any assignment, the value of an attribute (qualified or not) is undefined. Sometimes it is necessary to un-define a previously defined attribute value. This is particularly the case for qualified attributes representing a collection of objects, when an object is to be removed from that collection. For these cases, the un-assign statement `object1.attribute1[object2] :-` is provided.

MOL is untyped, so that attributes are defined on a per object basis. There is no mechanism of declaration and/or inheritance, and thus also no condition of illegal access. Due to the special properties of MOL discussed in the next section, access to undefined attributes can be caught prior to program execution.

Variables. As usual, MOL offers variables as named placeholders for objects. Variables need not be declared before they can be used; they are introduced on the fly.

Variables are assigned values through the assignment operator `:=`. Before its first assignment, the value of a variable is undefined; an un-assignment of a variable resets its value (to undefined).

Control flow. In MOL, all statements are executed sequentially. Branching and loops are enabled by the while statement, which loops through a sequence of statements as long as a given condition is satisfied. Since MOL has no built-in data types, the only possible conditions are tests for identity (==) and non-identity (<>) of two objects.

While the while control structure is sufficient to write arbitrary programs, pure while programs are difficult to read. Even though MOL is designed as an intermediate language not intended for source level programming, for the sake of our examples (and simplicity of mapping rules) we add the usual branching and loop control structures (which can be introduced as needed). Also, we add the possibility of iterating through the qualified attributes of an object; the housekeeping necessary to realize this feature (basically keeping track of the used qualifier objects) can be expressed in MOL, so that the expressiveness of the language is not affected.

Besides branching and looping, MOL allows the calling of subroutines, called *procedures*.

Programs and procedures. A MOL program is a set of procedures, where each procedure is a sequence of statements. A procedure has at least one parameter, the receiver, which is implicit and referred to as `self` from within the procedure. All parameters are treated as variables whose scope is restricted to the procedure (local variables). All other variables used in a procedure are global. Objects created in a procedure are introduced to the whole program, i.e., to all other procedures.

Since MOL is untyped, procedures are not associated with particular classes of objects. In fact, any procedure can be called on any object, and therefore cannot make any presumptions on the definedness of attributes (see above). Also, procedures – like attributes – are not inherited.

MOL is a procedural language. In particular, it does not have notions of events, message passing, or even concurrency. This restriction facilitates much of our theoretical reasoning carried out in the next section; it may be lifted in future versions of our work.

I/O and user interaction. Interpreted as a program, a model is special in that its communication with the outside world can be reduced to the communication with those evaluating the model, which will be its designers and their clients. Since the only user interaction that can be modelled in UML is that with an actor, it suffices that the executed model takes simple input (i.e., prompts the user to enter an object name which is then stored in a variable) from an input device, and displays its trace (or just certain states; analogous to the spy mechanism in PROLOG interpreters) on an output device. Note that by means of input no new names can be introduced to a program; nor can objects be created (or deleted).

Import of data types. Certain homogeneous sets of objects and functions defined on them—collectively known as data types—are naturally found in many modelling domains. Because it cannot be the purpose of modelling to define these objects or their functions, the objects may be assumed to be pre-existing and should not be created or destroyed by a program; likewise the functions are predefined and not modelled as object interactions. Data types are imported; by importing, their elements become valid object names and functions of the importing MOL program.

Data types need not be closed; for instance, a data type with the integers from 0 to 20 as its objects and addition as its function is acceptable; addition is then undefined for sums greater than 20. Note that data types do not introduce types to MOL through the back door; all objects of an imported data type become objects of the program on equal footing with those introduced by the program; functions imported with a data type are generally partial (in that they are not defined for all objects). To avoid infinite domains (and hence the loss of the properties of MOL described in Section 5), the objects of generic data types (such as number) must be limited to finite enumerations.

4.2 The syntax of MOL

The syntax of MOL is given by the following set of EBNF rules

```
Program      ::={ Import | Procedure }.
Import       ::="import" DataType ".".
Procedure    ::="procedure" ProcedureName "(" Parameters ")" Body ".".
Parameters   ::=[ VariableName { "," VariableName } ].
Body         ::=Statement | Statement ";" Body.
Statement    ::="new" ObjectName | "delete" ObjectName | Assignment
               | Unassignment | Input | WhileLoop | ProcedureCall.
Assignment   ::=Alias ":=" Referent.
Unassignment  ::=Alias ":-".
Referent     ::=ObjectName | Alias | “self”.
Alias        ::=VariableName
               | Referent { "." AttributeName "[" [ Referent ] "]" }
               | Referent { ".” FunctionName "(" Qualifiers ")" }.
Qualifiers   ::=[ Referent { "," Referent } ].
Input        ::="?" Alias.
WhileLoop    ::="while" Condition "do" "{" Body "}".
Condition    ::=Referent "==" Referent | Referent "<>" Referent.
ProcedureCall ::=[ Referent "." ] ProcedureName "(" Qualifiers ")".
```

where `DataType`, `ProcedureName`, `FunctionName`, `VariableName`, and `AttributeName` are identifiers; `ObjectName` stands for arbitrary contiguous (i.e., not white-space separated) strings, including numbers.

4.3 Operational semantics of MOL

All referents in a MOL program are evaluated to object names. A referent that is an object name needs not be evaluated (it evaluates to itself). A referent that is a variable evaluates to the name of the object that is the value of the variable. A referent of the form `<referent1>.attribute1[]` evaluates to the unqualified attribute value of `attribute1` applied to the object referenced by `<referent1>`, which is evaluated recursively. Accordingly, a referent of the form `<referent1>.attribute1[<referent2>]` evaluates to the attribute value of `attribute1` applied to the object referenced by `<referent1>`, qualified by `<referent2>`, which are both evaluated recursively. Lastly, a referent of the form `<referent1>.function1(<referent2>)` is evaluated by first evaluating `<referent1>` and `<referent2>` and then fetching the function value of `function1` as applied to the results of evaluation from the data type to which the object name of `<referent1>` belongs.

A condition of the form `<name1> == <name2>` where `<name1>` and `<name2>` stand for object names, evaluates to true if and only if `<name1>` and `<name2>` are identical. A condition of the form `<referent1> == <referent2>` evaluates to true if and only if the referents evaluate to identical object names.

Assignment of a variable is interpreted as usual, i.e., immediately after executing `v1 := o1`, `v1` = `o1` (where = is metasyntactical equality) and the conditions `v1 == o1` and `o1 == v1` evaluate to true. Assignment of an attribute is interpreted accordingly, i.e., immediately after executing `o1.a[] := o2`, `o1.a[] = o2`. Assignments with a qualifying object, i.e., assignments of the form `o1.a[o3] := o4` leave the attribute values for all other qualifying objects unaffected. In particular, after the above assignment still `o1.a[] = o2`.

A sequence of statements of the form `S1; S2` is interpreted as the execution of first `S1` and then `S2`. A while statement of the form `while <C> do <B> end` where `<C>` is a condition and `<B>` is a block of statements means that as long as `<C>` evaluates to true, `<B>` is repeatedly executed (re-evaluating `<C>` after each execution).

A procedure call is executed by first evaluating the referents (actual parameters) of the call, if any, and assigning them to the variables (formal parameters) of the procedure head and then executing the statements in the procedure body.

The import of a data type introduces all names known by the data type to the set of object names known to a program. Note that the data type needs not define these names extensionally; it merely has to offer a procedure deciding whether a given identifier is a name of an object of the data type. Therefore, each data type can decide whether an object name represents an object covered by the data type; also, it can deliver the function values for all functions defined by the data type. The interface of a data type as seen from the standpoint of specifying the operational semantics of MOL consists of two operations: `knows(aName)` and `evaluate(aFunction, ParameterList)`.

4.4 Denotational semantics of MOL

Because of the simple design of MOL, its denotational semantics is straightforward and defined as usual: objects are mapped to constants from some semantic domain, attributes are mapped to functions, and so on.

5 Theoretical properties of MOL

MOL has variables, assignment, a sequential execution order, the `while` control structure, and standard input/output conventions. Had it also an arbitrarily large range of integers, it would be equivalent to the *while*-language [24, 34] and thus Turing equivalent, meaning that any computable function could be computed in MOL. However, in modelling we are not interested in the computation of functions, but in the design of the interplay of a structured set of objects (mostly non-numerical) and its effects in terms of changes in structure (or state). Consequently, MOL goes beyond the *while*-language by introducing general objects (not only integers) and providing for their interconnection via attrib-

utes. On the other hand, it restricts the number of objects to the number of names in a program. As a consequence, MOL is not Turing equivalent.

5.1 Computational power

Although the restrictions in MOL may appear unnecessary at first glance, they buy us certain formal properties of models that are actually very useful in a modelling context, namely they make many questions decidable (mostly even with efficient algorithms) that for general computability notions such as Turing machines or while-programs (as well as for a number of restricted models, e.g. loop-programs) are undecidable.

To examine the computational power of MOL, we first consider the restriction of the language that does not allow procedure calls. A state of such a MOL program *P* is given by values for all attributes and variables, plus a program counter. Since the number of objects created during execution of *P* is finite as well as the number of attributes and values, we conclude that the number *N* of states of *P* is finite. More precisely, let *n* denote the number of objects in *P*; let *s* be the number of statements in *P*; let *v* be the number of variables in *P*; let *Att* be the set of attributes in *P*, and for $f \in Att$ let ar(*f*) denote the arity of *f* (here limited to 2, since every attribute is a function of an object plus, optionally, one qualifier). Then the number *N* of states is bounded above by $s \cdot n^v \cdot \prod_{f \in Att} n^{\mathrm{ar}(f)}$.

Now execution of *P* can directly be simulated by a deterministic finite automaton *M*: The state set of *M* is the set of states of *P*. The input alphabet of *M* is the set of objects in *P*. Execution of an assignment, unassignment, new or delete statement translates directly to a state change of *M* (only modifying the (un-)assigned variable or attribute value), without consuming an input symbol (this is, in automata-theoretic terms, an ε-move). The input statement translates to a state change that modifies only the value of the read variable. The `while` statement is resolved in the known manner.

If we now allow procedure calls but disallow (direct or indirect) recursion, we see that, though every procedure call introduces new local variables, the number of active variables is finite at every state of the execution of *P* and thus, we still enjoy the property that the number of states is finite. We conclude that, again, such a program can be translated to a finite automaton. The number of states *N* then is the same as above multiplied by a constant, the number of nested procedure calls that can be active at the same time, which is bounded by the number of procedures in *P*.

Finally, if we consider MOL programs without further restrictions, i.e., allowing arbitrary procedure calls, it is clear that the number of variables that are active at a time is unbounded, because in general we can no longer bound the recursion depth of an execution of *P*. The number *N* of states thus can no longer be bounded as above, but depends on the possible recursion depth which, in turn, depends on the input from the user. To simulate *P*, we now need the power of deterministic pushdown-automata [21, pp. 246ff]. Program *P* is translated into such an automaton *M* as above, with the addi-

tional rule that for procedure calls, all local variables are saved on the stack and restored upon return from the called procedure. All other MOL statements lead to the same state changes as described above for programs without procedure calls.

We remark that the just given simulation results also hold vice versa since, easily, every finite automaton can be simulated by a non-recursive MOL program with only one procedure and every pushdown-automaton can be simulated by a MOL program with procedure calls. Hence, the computational power of MOL is exactly that of finite automata or pushdown automata, depending on the absence or presence of recursion.

5.2 Reachability properties

Today's large applications are best viewed as cohorts of (possibly remotely located) interacting objects, exchanging information and relying on each other's services. The question in modelling is then not so much whether a given (overall) function is computable, but more the verification of protocols, i.e., whether the objects cooperate as intended by the model specification, and whether the cooperation comes to an end (so that the result can be delivered). In theoretical computer science, this is known as the *halting problem*. The question if a MOL program *P* halts given a fixed input sequence reduces to the question if the corresponding automaton reaches a certain state (or any of a certain set of states). Stated differently, *P* does not halt if the corresponding computation of *M* is caught in an ε-loop. This is a graph reachability problem and can be solved efficiently using, e. g., Dijkstra's algorithm that has a run-time of $O(N^2)$, where *N* is the number of nodes of the graph [15, pp. 13–15], i.e., the number of states of the MOL program as calculated above.

Further reachability type questions can be answered again in time $O(N^2)$ using the same algorithm, e.g., if a certain state will ever be reached, or if a certain state will always be reached before a second state (for instance, if an object is created or a variable is initialized before it is used). The latter in particular grants us the freedom to live without `null` or `nil` as a special value, usually needed to turn *undefined* into a defined condition.

5.3 Equivalence of models

Modelling is a dynamic activity that, in real projects, never ends. As models evolve, they become muddled up so that, in order to maintain their clarity, they have to be refactored. Since it is the nature of a refactoring that it leaves the observable behaviour of the refactored artefact unaffected, refactoring a model means that the model before and after the refactoring must be equivalent [44]. An automatic equivalence check is thus highly desirable.

The above described connections to different types of automata provide such a check: Say that two automata M_1 and M_2 (either finite automata or pushdown-automata) are *equivalent* if for every input sequence, both M_1 and M_2 end in an accepting state or both end in a rejecting state. It is known that equivalence of deterministic finite automata can be checked in time $O(N^2)$, where N is the number of states in the given automata. Concerning deterministic pushdown automata, G. Sénizergues has only recently shown that equivalence is decidable [35]. However, he does not give an explicit time-bound of the decision algorithm. That there is an algorithm that has a primitive-recursive time-bound was shown in [43]; this means in other words that the algorithm can be implemented using loop-programs.

The just mentioned notion of equivalence of automata translates to MOL programs in the following way: Two MOL programs are equivalent if for every sequence of input symbols, both end halting or both do not (yet) end halting. This notion of equivalence may be sufficient for certain restricted areas, but in general something more involved is needed. As argued in the preceding sections, in object-oriented models an interaction with the user will lead to some final object structure, i.e., some set of existing objects with interrelations (in the form of attributes) between objects. Thus, the interaction with the user should be considered as input, and the produced object structure should be considered as result of the computation. Two models or two MOL programs should be considered equivalent if for every input sequence they produce the same object structure. On the level of automata, this corresponds to *token automata*, i.e., usual (finite or pushdown) automata that produce a symbol from an output alphabet in the end of their computation, i.e., in their final states. The well-known equivalence test for finite automata can be adapted to yield an equivalence test for token automata: one only has to differentiate between final states that produce different output symbols (see [18]). The run-time of the extended algorithm is again $O(N^2)$ for automata with N states.

We have implemented this kind of equivalence check in our tool set (see Section 7). Our prover is thus able to decide if two given UML models are equivalent in the sense that the same interaction with the user results in the same object structure.

A third, stricter notion of equivalence would require that for every input sequence, the two examined MOL programs produce the same trace. On the level of automata, two programs might be defined to be equivalent if for all inputs they lead to the same traces. This has to be modelled by transducers (automata with output), i.e. finite transducers and deterministic pushdown transducers. We only mention in passing that also in this setting, equivalence remains decidable [26, 36], in the case of finite transducers with a decision algorithm with run-time $O(N^2)$ [4].

The need for checking the equivalence of two models may also arise in the early phase of model construction, if different alternatives for the solution of a modelling problem are to be explored. While equivalence of models is an interesting question, the automatic identification of the (semantic) difference between two models may be even more interesting. The theory of formal languages provides an answer for the special case of diagrams leading to MOL programs without recursion—thus,

again, we deal with finite automata. Given two finite automata M_1 and M_2, it is possible to determine efficiently the set of all words for which M_1 and M_2 yield different outputs; formally this is the symmetric difference $L(M_1) \Delta L(M_2)$, where $L(M_i)$ is the set of words accepted by M_i, $i = 1, 2$, and for any sets S, T, $S \Delta T = (S \cap \neg T) \cup (T \cap \neg S)$.

In fact, $L(M_1) \Delta L(M_2)$ is known to be always a regular set [21, pp. 131ff]. A corresponding finite automaton that accepts exactly these words can be constructed in time $O(N_1 \cdot N_2 \cdot n)$, where N_i is the number of states of M_i, $i = 1, 2$, and n is the number of input symbols (in the MOL context: the number of objects). This follows from the facts that an automaton for $L(M_1) \cup L(M_2)$ can be constructed in time $O(N_1 \cdot N_2 \cdot n)$ [45, p. 117] and that an automaton for $\neg L(M_1)$ can be constructed in time $O(N_1)$ [45, p. 117]. Thus, given an input sequence w, if w is handled differently in two UML models, i.e., leads to different outputs of M_1 and M_2, can be decided in time $O(N_1 \cdot N_2 \cdot n)$. Note that the complexity bound does not depend on the length of the input to the automaton, $|w|$, that is in our context, the length of the interaction with the user.

5.4 Efficiency considerations

The previous subsections showed that a number of problems become decidable for MOL programs that in general and even in the case of restricted computation models (such as loop-programs) are undecidable. Note that the equivalence check for nondeterministic pushdown automata is already undecidable [21, p. 407f]; hence the constructs in the MOL in a sense touch the border of what can be done in principle.

To appreciate the practical usefulness of our approach it is important to know that for MOL programs that are non-recursive (and object interactions are typically non-recursive) the given algorithms are very efficient: they have a run-time bounded by a polynomial (of small degree, mostly linear or quadratic) in the number of states, which in turn is kept small typically since the number of objects of a UML model is small (in the order of a few hundreds).

6 Mapping UML-A diagrams to MOL programs

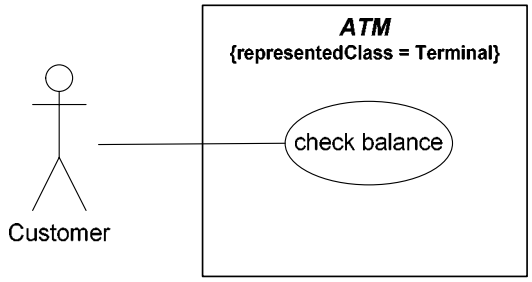
Single diagrams show certain views of a system. Although these views need not generally be isolated from each other, their integration is usually left to the creativity of the reader, without strict binding rules (comparable to those of a linker linking different modules of a program) being given. If the automatic integration of different diagrams into a single program is the goal (as is in our case), such binding rules are indispensable; they must be part of the language specification.

It follows that UML-A cannot be a pure subset of UML, because UML lacks such rules. What we propose instead is a graphical modelling language that is derived from UML, with its ambiguity taken

out and certain elements of co-reference (which are required for giving models a program semantics) added. All this comes of course at the price of flexibility. However, making UML-A so general that the existing UML definition can be built on it should compensate for this.

6.1 Use case diagrams

Use case diagrams specify interactions between outside actors and the modelled system. The outside actor corresponds to a role of a user (be it human or another system); during model validation, this role may be assumed to be filled by a validator, i.e., a stakeholder of the model. Although actors are outside the modelled system, their roles are not: the system associates with each role a set of services it has to offer, and the runtime of model execution must make provisions for a user to log on in one of the offered roles and select associated services for validation.




```
ATM_check_balance(customer)
  ? customer.terminal[];
  ? customer.account[];
  ? customer.pin[];
  terminal.Terminal_check_balance(
    customer.account[],
    customer.pin[]
  )
.
```

Figure 2. A use case diagram and its translation to MOL code (simplified). The procedure is called by the runtime system of the MOL interpreter after a user has logged on in the role of a customer and selected the use case for validation (execution). The requested user input and nested procedure call have been derived from information provided in the collaboration diagram of Figure 5.

The system itself (in a use case diagram depicted as a box surrounding the use cases) is an abstraction offering the services provided by its components. As indicated by our use of the term component, we interpret the underlying abstraction principle as that of composition; in fact, we assume that there is some (abstract; *cf.* [40]) class that represents the system, whose instances are composites made up of others. Thus, should there be a class symbol in some other diagram that has the same name as the system, we interpret this as a sign of co-reference, i.e., that class and system are the same entity, offering the same functions. Should such a class be lacking, a use case can still be resolved, assuming that the abstract system has been decomposed into collaborating objects without further notice.

Since use cases represent the functions of a system, we interpret them as methods. Consequently, our model compiler tries to bind use cases to methods of the same name, preferably those defined in a class with the system's name. If no such class exists, it is checked whether there is an interaction diagram with an initial method of the use case's name. If it does not exist, the use case remains uninterpreted. If it does, the compiler extracts the parameters of the linked method and inserts input statements letting the user select objects (as actual parameters) in response to activating a use case. De-

pendencies among use cases give hints on the call dependency of the corresponding methods; the compiler could check their consistency with the expansion of a use case in an interaction diagram, but due to the fuzziness of UML's various use case expansion mechanisms it does not currently do so.

In a way, the set of all use case diagrams of a model represents the menu system of the system being modelled; this menu system is then specific to the role in which a user has logged on. It can accommodate both interpreted and uninterpreted system functions.

6.2 Class diagrams

A class diagram captures structural (type and other) constraints all states and scenarios of a model must comply with. While these constraints can direct and control the translation process from UML-A diagrams to MOL programs, they do not themselves end up in MOL statements: they are either used by the compiler to generate decision tables (for instance to resolve the dynamic binding of procedure calls), result in constraints that are enforced during the execution of a model (e.g., multiplicities), or are simply compiled away (e.g., after it has been proven that a model is statically type correct). Since enforcing type constraints requires commitment to a certain type system (which is lacking in the UML standard), we perform only the most fundamental checks; we argue that this is in line with the spirit of UML, which uses types mostly for data modelling (the class diagram is really a derivative of the entity-relationship diagram, and collaboration diagrams use a very loose form of types) and not for making a model safe. Indeed, as noted in [31], untyped languages are preferred during the early, prototyping stages of system development, whereas strict typing is usually only appreciated at later stages, because of their potential to introduce correctness and efficiency.

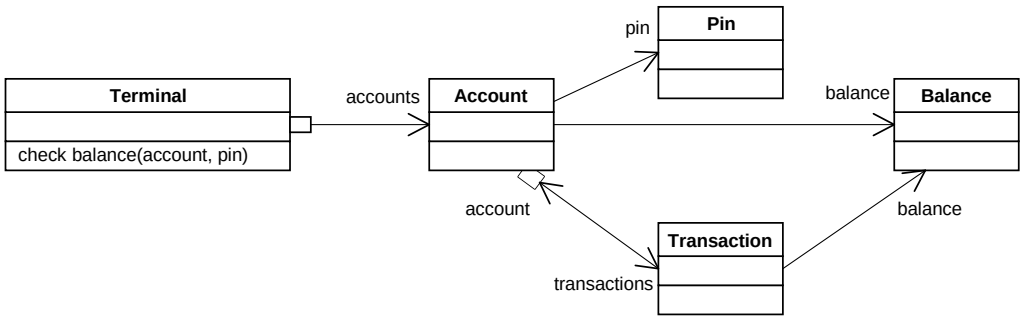


Figure 3. Class diagram of an ATM.

In our current implementation of the compiler, class diagrams are used only to guide the translation process [20]. Future versions will include the insertion of invariants such as multiplicities in the generated MOL code.

6.3 Object diagrams

With class diagrams being compiled away, object diagrams are responsible for injecting object structures into a model. Such object structures are needed as the starting points for use cases; hardly any use case will start from scratch by creating the objects it needs explicitly. Object diagrams can also serve to specify the outcome of a use case; as such, they must be interpreted as postconditions for the corresponding operation.

Since all methods operate on object structures, execution of a method as specified in a collaboration diagram (see below) requires a set of objects interlinked to reflect the associations required for the collaboration. Unless the collaboration diagram is ground, such a structure must be created prior to execution (as creation is not part of the collaboration), and this is done by the modeller providing object diagrams. Since there may be several alternative object diagrams suiting a single collaboration diagram, the compiler only needs to check the availability of such diagrams, and can leave the choice to the user at runtime. Should there be no such diagrams, the corresponding method cannot be executed. It is however possible that a suitable object structure is created dynamically (as the result of some other use case or method), so that the compiler does not need to check the availability of fitting object diagrams at all; it follows that a use case may remain unexecutable, due to the unavailability of a suitable object structure. To avoid these issues, we currently assign (through a corresponding stereotype and tagged values checked by the model compiler) to each use case one object diagram serving as its initialization (not shown in Figure 2).

An object diagram is translated as follows. All named objects are translated to a `new` statement followed by the object's name. Note that if the object already exists (because it has been created through the execution of another object diagram), repeated creation remains without consequences.

A pair of objects `(o1, o2)` connected by a link `l` maps to

- an assignment of the form `o1.l[] := o2` if `l` is directed from `o1` to `o2` and the only link with name `l` starting from `o1`; or
- an assignment of the form `o1.l[o2] := o2` if `l` is directed from `o1` to `o2` and there are several (unqualified) links with name `l` starting from `o1`.

If the link is bidirectional, the opposite directions are mapped accordingly.

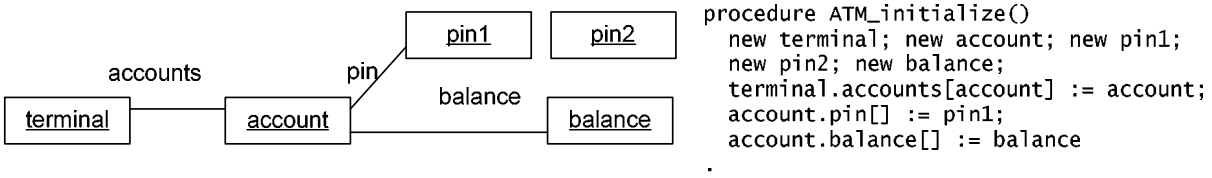


Figure 4. Object diagram specifying an initial structure for the use case of Figure 2. Class names have been omitted for the sake of brevity. The second pin is provided so that the user can make a false input.

Because of the open world assumption of modelling, that a link is not shown does not mean that it is not there. This is particularly true for the peripheral objects of a diagram, whose specification may be continued in another. However, the fact that an object occurs in several object diagrams (co-referenced) does not necessarily imply that the corresponding generative MOL code is also executed; instead, different object diagrams might describe different constellations at different times. As mentioned above, we currently solve this problem by tying exactly one object diagram to each use case diagram; future versions of our work might allow the definition of different scenarios (with all diagrams of a single scenario complementing each other) and have the user select a scenario dynamically, providing the initial system state for a use case to be executed.

6.4 Sequence diagrams

Whereas use case diagrams can, but need not show the decomposition of a method, it is the purpose of a sequence diagram to show how an operation is performed by breaking it down to others. However, a sequence diagram is more than a function tree or a flowchart: it associates with each operation a possibly different object (either a named one or a placeholder). This is in contrast to the use case diagram, in which the enclosing object, the system, is the same for all use cases. In fact, as has been noted elsewhere, the sequence diagram integrates functional with structural decomposition [39]: along with the operation specified, the whole (the system) is decomposed into a set of collaborating objects. Therefore, the sequence diagram is central to model integration.

Unlike for the related collaboration diagrams, the focus of sequence diagrams is on showing the flow of interactions between objects. If interaction corresponds to method calling, sequential interaction corresponds to a sequence of method calls. Branching and loops, which may also be expressed in a sequence diagram, are translated to the corresponding MOL control structures. Asynchronous communication and active objects however must be mapped to parallel processing, an issue not considered here (*cf.* Section 4.1); the necessary synchronization of processes would confront modelling with low level issues from which it tries to abstract, contradicting its very purpose.

A sequence diagram is bound to a use case if the initiating method of the sequence diagram matches to the operation name of the use case, and if the invoking actors are compatible. Note that this includes the case that the same operation is specified differently for different actors; the use case

must then either specify all actors, or comprise them under a suitable generalization. It is important to note that the system and the class of the object offering the operation need not match: since the system is considered an abstraction of its components, which are collaborating objects, the sequence diagram shows the collaboration, regarding the system as an abstract instance, i.e., an instance that disappears upon zooming into the model [39].

6.5 Collaboration diagrams

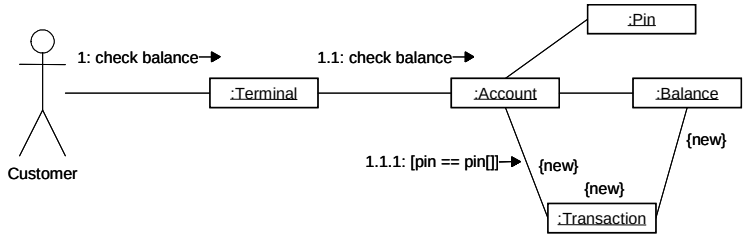
Collaboration diagrams add to sequence diagrams the links between objects (or their placeholders) necessary to exchange messages. These links express the knowledge objects have of each other, initially and as it changes during the course (and as the effect) of the collaboration. The sequencing of method sends is captured in the numbering of calls; control structures and their mapping to MOL are analogous to that of sequence diagrams, as is everything else.

Collaboration diagrams can be specified on the instance and on the specification level. Specifications on the instance level are based on object diagrams; they add to them the message sends, plus stereotypes or constraints hinting at newly created and deleted objects and links. They can be considered instantiations of diagrams on the specification level, in which objects are represented through placeholders.

Ground collaboration diagrams show the collaboration for specific objects, suggesting that the same operation would result in a different collaboration for others. Although such alternative collaborations can be provided to cover alternative paths of performing the same operation, one usually strives for generic descriptions covering as many different instantiations as possible. Such generic descriptions invariably involve placeholders—they are made possible by collaboration diagrams on the specification level. Note how this closely corresponds to the use of variables in programs: the more variables it contains, the more compact the specification gets, and variables can be eliminated by rolling out the code in their scope. In fact, collaborations on the specification level closely correspond to procedures of a MOL program; this will be exploited in the translation process.

In order for a collaboration diagram to be executable, the following conditions must be satisfied. It must specify an entry point, either in the form of an actor calling a method or some other ingoing message call not issued by an object of the diagram. If the method is called by an actor, method and actor must correspond to a use case and an actor of the same type (or a supertype thereof), respectively, in a use case diagram. Before the operation can begin, an object structure matching the initial structure of the collaboration diagram must be identified. This structure should either be provided and/or selected by the user (who, in the role of the actor, started the operation) or resultant from the execution of a previous operation. Note that the object structures must comprise the links required by nested operations, i.e., by operations specified in different collaboration diagrams, but invoked by the original col-

laboration. Because of the weak typing of collaboration diagrams and their connection to class diagrams, we assume it suffices that a correctly typed object diagram exists; no additional type checks are performed by the compiler.




```
procedure Terminal_check_balance(account, pin)
  self.accounts[account].Account_check_balance(pin)
.
procedure Account_check_balance(pin)
  if pin == self.pin[] then {
    new transaction;
    self.transactions[transaction] := transaction;
    transaction.balance[] := self.balance[];
    transaction.account[] := self
  }
.
```

Figure 5. Collaboration diagram showing the interaction between customer, terminal, and account. If entered and stored pin are identical, a new transaction object is created and linked to the account and balance. Parameters and receiver of the initial call are extracted and inserted as inputs to the translation of the use case of Figure 2. Note that the navigability of the new links must be derived from the class diagram (Figure 3).

Figure 5 shows the collaboration diagram for the use case of Figure 2 and its translation to MOL code. The translation is straightforward: it creates two procedures, one associated to class `Terminal`, the other to `Account`. There is no result returned to the customer; according to our understanding, the changed object structure is the output of the executed model (recall that MOL has no output statement). Note the overloading and its resolution through the renaming of methods. Since objects are untyped, MOL in its initial version has no notion of dynamic binding.

6.6 Statecharts

As noted elsewhere [39], the states of a statechart are only loosely linked to the states of an object as expressed by its attribute values and by its links. In fact, since MOL has no notion of state separate from the linking of objects through attributes, translation of statecharts to MOL programs is impossible as long as this mapping remains unspecified. Note, however, that certain proposals of executable UML address this problem by attaching procedures to the entry actions of a state, and by accessing (possibly changing) attribute values and links in these procedures [25]. However, this convention does not ensure that different states of the statechart are reflected in different attribute values and links of the object: an empty entry action for example leaves the values unaltered.

6.7 Activity diagrams

If interpreted as an object-oriented variant of flowcharts, the translation of activity diagrams to MOL programs should be straightforward. For instance, decisions map to branches, subactivity states map to procedure calls, and a whole activity diagram maps to the body of a procedure. Action states and call states would map to primitive operations, but since the only primitive operation of MOL is assignment, they may alternatively entail user interaction of some kind (e.g., printing the name of the state on the console so that the user knows which state is being "executed"). Object flows map to parameters of procedures, signal sending to asynchronous procedure invocation.

Because the exact meaning of activity diagrams is difficult to grasp from the standard and it appears that in practice they are used rather liberally, we defer precise translation rules to a later stage, speculating that the formal semantics of activity diagrams could well be influenced by the precise definition of translation rules to MOL programs.

7 Tool support, process model of language design, and example of use

Language design is inherently difficult. Linguistics teaches us that it is less of a creative than an evolutionary process: like with natural languages, the success of any artificial language depends on its usability, and if the language fails to adapt to the users' needs, it will not be used.

The usability of formal languages critically depends on the existence of tools realizing their definition. If such tools lack, the formal language will quickly be used informally, subjecting it to the much criticized ambiguity of natural language, which it was to avoid. In fact, only tools watching over the strict use of the language can tell us whether it is adequate for the chosen purpose, and which its flaws are. This insight guides our development process for UML-A.

7.1 The iMOL Framework

We have developed and implemented a first version of a model compiler (from UML-A to MOL) translating the XMI output of a commercial UML editor to MOL programs [20] (see Figure 6). The compiler enforces certain annotations to models that guide model integration; it also has some other peculiarities not discussed here.

MOL programs can be entered, executed, and debugged in an integrated development environment called iMOL [5]. iMOL can start the model compiler and take over its output; it also incorporates the runtime functionality of model execution, i.e., the menu system and all other routines necessary for user interaction (such as selecting objects as actual parameters).

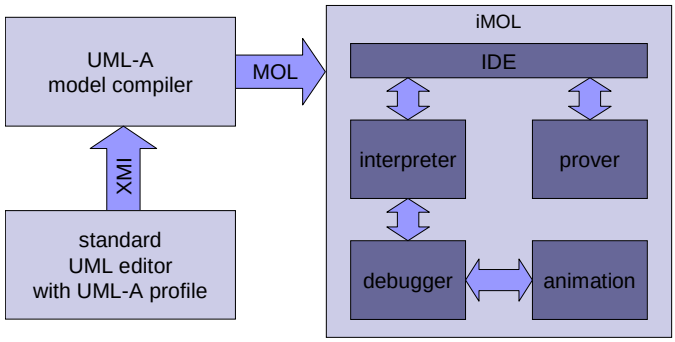


Figure 6. Tool support for the evaluation of UML-A and MOL.

Rather than being part of the model compiler, iMOL is designed as a stand-alone tool that allows us to explore MOL in practice and learn about its deficiencies. Changes to MOL are first integrated in iMOL and tried out in isolation; only if the changes make sense (and if they affect the translation process from UML-A or its definition), we move them to the model compiler. We chose this reverse engineering approach because we reckon that we have a rather good understanding of what makes a program, but are yet unaware of many of the subtler model integration problems.

The animation unit of iMOL allows interactive graphical model execution and serves as the user interface for model validation. It has been complemented by an off-line visual renderer automatically transforming program traces (obtained via iMOL's debug interface) into high end 3D animations that can be used as teaching material or as proposals to project management [38]. A screenshot from the ATM example as used in Section 6 is shown in Figure 7.

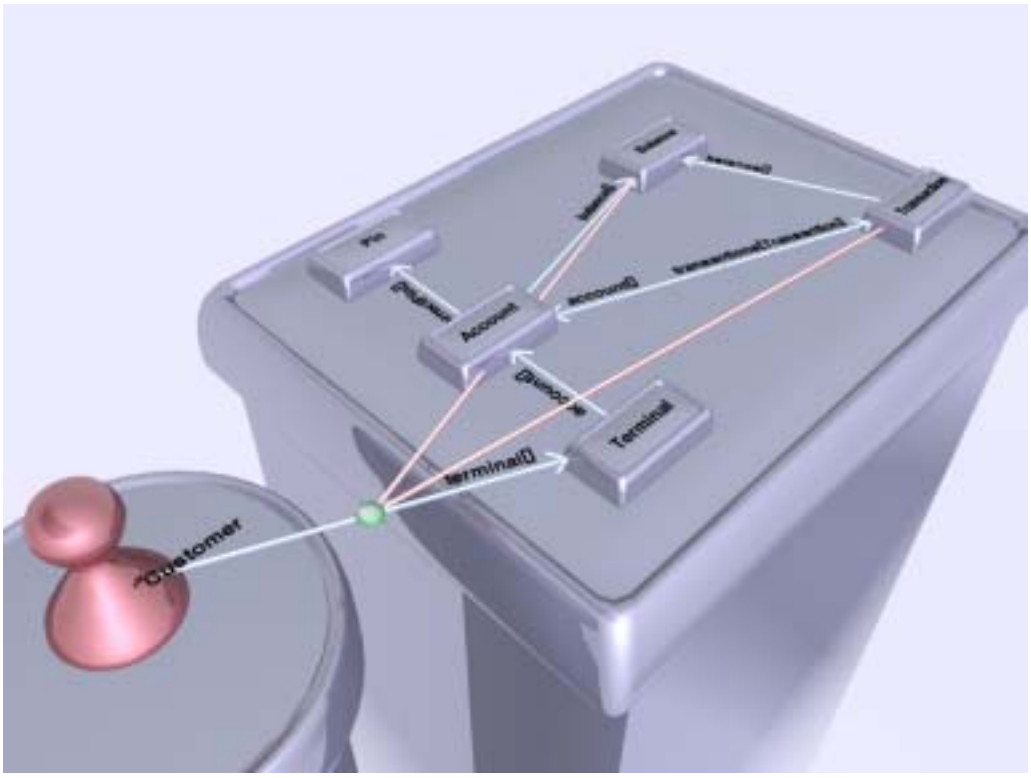


Figure 7. Snapshot from a high-quality animation automatically generated from the execution of the model of the ATM example [38]

The prover unit currently implements the equivalence test of token automata described in Section 5, i.e., the prover is capable to determine if two given models are equivalent in the sense that for every possible interaction with the user they produce the same object structure [1]. To give an impression of its capabilities, we have modelled two alternative solutions to a standard problem from the logic domain. The two versions stand for a single model before and after a rather complex refactoring. The results can of course be transferred to any other domain, but examples of less academic refactorings will typically require a broader setting than can be reproduced here.

7.2 iMOL at Work

A common problem of mathematical logic is to decide whether a given set of Boolean clauses is satisfiable, i.e., whether or not there exists a set of variable assignments that lets each clause in the set evaluate to true. Since a clause is a set of literals (negated or non-negated Boolean variables) joined by disjunction, it suffices that one literal of each clause evaluates to true. The problem is of general interest because any formula stated in propositional logic can be transformed into such clause form. In theoretical computer science, the problem is known as SAT and is the prototypical (and also historically first) NP-complete problem.

For our example, we consider a set with three clauses, $C1$ through $C3$, with a total of six literals, $L1$ through $L6$, such that

$$C1 = \{L1, L2\},\ C2 = \{L3, L4\},\ C3 = \{L5, L6\}.$$

The odd numbered literals are all bound to one atom (Boolean variable), *A*, whereas the even-numbered literals are bound to another, named *B*. User input determines which of the literals are negated, and which are not.

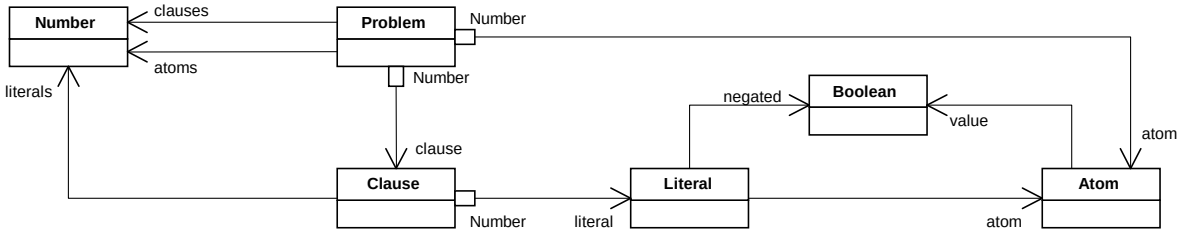


Figure 8. Class diagram of the SAT problem.

Figure 8 presents a class diagram of the given problem. `Boolean` and `Number` are imported data types with predefined functions. Figure 10 shows the object diagram representing the initial configuration, i.e., the construction of the clause set. The `negated` links of the literals are still undefined; corresponding objects (`true` or `false`) must be selected by the user as input once the use case `solve` is selected for execution. Note that, deviating from Section 6.1, the input variables are not determined by the formal parameters of the initial method, but are marked explicitly (by question marks in the model); this is to make the initiating procedure independent from the initial configuration, which de-

termines the number of literals and thus, also, the number of inputs. Quite clearly, the number of objects in the example is finite, the example thus satisfying the finiteness requirement of our formal framework.

The solution of the SAT problem is triggered by calling the `solve()` procedure. That `solve` is an entry procedure selectable by the user of iMOL is defined by the use case diagram shown in Figure 9. The `representedClass` constraint tags the use case `solve` to class `Problem`, which has a corresponding method. The MOL compiler takes this information to generate two default objects, `user` and `system`, and to call the initialisation method generated from the initial object diagram (Figure 10) on `system`.

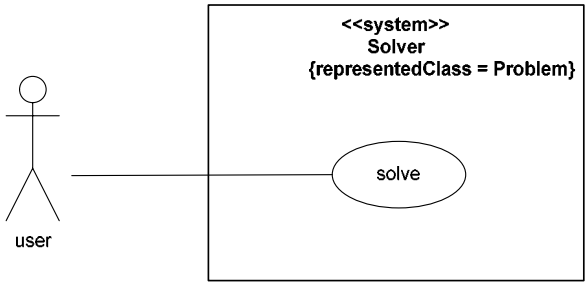


Figure 9. Use case diagram declaring an initiating method decomposed by a corresponding implementation in class `Problem`.

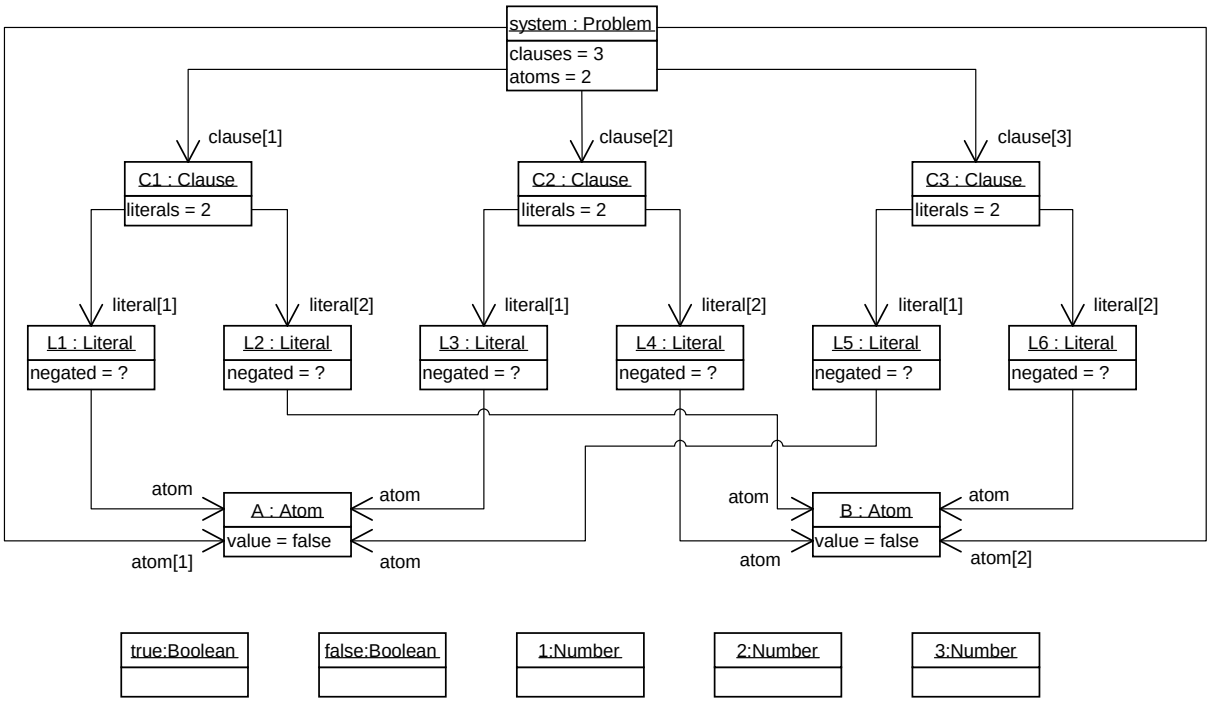


Figure 10. Initial object diagram. Links to numbers and truth values are represented as attributes, which is a shorthand notation for links (*cf.* Sections3.2 and 4.1).

An obvious solution to the SAT problem is shown in Figure 11. It evaluates the clause set for each possible assignment of truth values to the atoms *A* and *B*. For this purpose, a few additional (unquali-

fied) attributes (links) are needed; these are `solved` (for the problem and the clauses) and `finished` (for the problem).

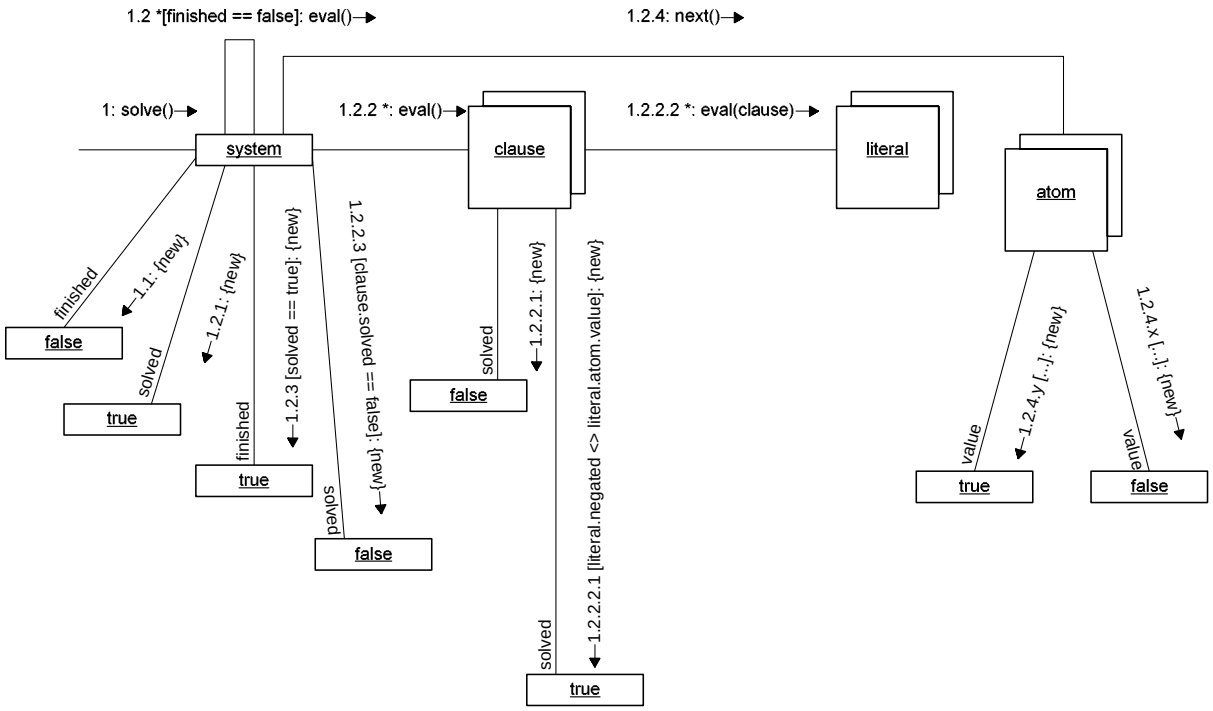


Figure 11. Collaboration diagram of Version 1. A message with a `{new}` constraint establishes a new link to the receiver object. It represents an assignment.

The solution procedure is straightforward: a clause evaluates to true if and only if the value of one of its literals' atoms is true and the literal is not negated, or if the literal is negated and the atom's value is false (i.e., `literal.atom[].value[] <> literal.negated[]`). This condition is checked for all clauses in the problem, with the evaluation accumulated in `clause.solved` and `problem.solved`, respectively. The procedure generating the truth assignments, `next()`, is recursive and therefore not modelled here (*cf.* discussion in Section 3.3); its MOL implementation is included in Figure 13. Once a solution has been found, `problem.finished` is set to `true` and the procedure terminates.

The idea of the second, refactored solution to the SAT problem is that in order to satisfy a clause set, it suffices to select one literal from each clause such that all selected literals can simultaneously (i.e., under the same variable assignment) evaluate to true. This is the case if and only if the selection is free of contradictions, i.e., contains no negated and non-negated literal of the same atom. The algorithm thus cycles through all possibilities of choosing either the first or the second literal from each clause, and then tests for a contradiction in each pair. Interestingly, the search for a solution does not touch atoms *A* or *B*; also, it does not need to memorize the solved state of individual clauses, but instead has to keep track of the literals it visits in each clause. Thus, it needs different helper attributes (`pair` and `try`).

Figure 12 presents the collaboration diagram for the alternative solution procedure. It requires a slightly modified initial state, which has been omitted here for the sake of brevity (the necessary changes can be deduced from Figure 13). Its equivalence (as regards observable behaviour, i.e., effect) with that of Figure 11 is unobvious, so that a formal proof would seem desirable. With iMOL, such a proof can be delivered by translating the models into MOL programs and checking these for equivalence. The MOL code corresponding to the two models is shown in Figure 13; as with Version 1, the `next()` procedure generating all selections is recursive and has not been worked out in the collaboration diagram.

Starting iMOL's equivalence check on the two programs unveils a practical problem with our approach. Due to the untypedness of objects and variables, the user can assign any object to the input variables, even though only `true` and `false` are meaningful. While this does not affect equivalence, practically it means that the corresponding test has to check all, including meaningless, inputs. Even though the cost of equivalence testing is only quadratic in the number of states (Section 5.3), the number of states grows – in the worst case – exponentially with the number of objects (base) and the number of variables and attributes (exponent) (Section 5.1). While the program flow usually constrains this growth dramatically (only objects that can actually be assigned to a variable lead to new states), user input is generally unconstrained. Thus, the 17 objects of Figure 10 and its 6 inputs boost up the number of states by a factor of approx. 24 million (as compared to a factor of 64 if only Boolean inputs were allowed). This growth would be naturally cut back by typing, but for reasons given above MOL is untyped. As a workaround, the following helper procedure

```
procedure input(literal)
    literal.negated[] := false;
    ? input;
    if input == true then
        {literal.negated[] := true};
    input :-.
```

ensuring that the input is always meaningful does the same: it reduces the number of states from a theoretical maximum of approx 3.4 billion to 8934 for Version 1 and from approx. 2.6 billion to 6822 for Version 2. Of course, explicit validity checks of this kind are precisely what we were trying to avoid for modelling by not including `null` or `nil` values; the good news is that guarding input variables as above is all that needs to be done to reduce complexity: typing of all variables and objects would not reduce complexity further [1].

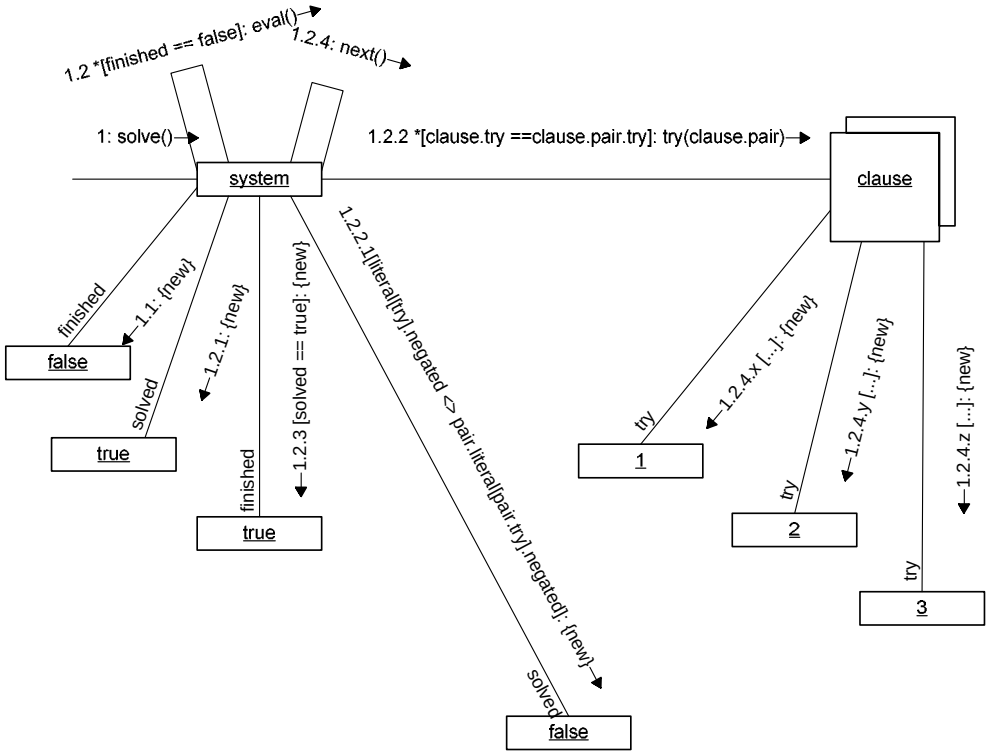


Figure 12. Collaboration diagram of Version 2.

```
import Boolean.
import Number.

procedure start_Problem_solve()
  new A; new B;
  new C1; new C2; new C3;
  new L1; new L2; new L3;
  new L4; new L5; new L6;
  system.atom[1] := A;
  system.atom[2] := B;
  system.atoms[] := 2;
  system.clause[1] := C1;
  system.clause[2] := C2;
  system.clause[3] := C3;
  system.clauses[] := 3;
  C1.literal[1] := L1;
  C1.literal[2] := L2;
  C2.literal[1] := L3;
  C2.literal[2] := L4;
  C3.literal[1] := L5;
  C3.literal[2] := L6;
  L1.atom[] := A;
  ? L1.negated[];
  L2.atom[] := B;
  ? L2.negated[];
  L3.atom[] := A;
  ? L3.negated[];
  L4.atom[] := B;
  ? L4.negated[];
  L5.atom[] := A;
  ? L5.negated[];
  L6.atom[] := B;
  ? L6.negated[];
  A.value[] := false;
  B.value[] := false;
  system.Problem_solve().

procedure Problem_solve()
  self.finished[] := false;
  while self.finished[] == false do
    {self.Problem_eval()}.

procedure Problem_eval()
  self.solved[] := true;
  foreach clause in self.clause do
    {self.clause[clause].Clause_eval()};
  if self.solved[] == true then
    {self.finished[] := true};
  self.Problem_next().

procedure Clause_eval()
  self.solved[] := false;
  foreach literal in self.literal do
   {self.literal[literal].Literal_eval(self)};
  if self.solved[] == false then
    {system.solved[] := false};
  self.solved[] :-.

procedure Literal_eval(clause)
  if self.negated[] <> self.atom[].value[]
                                          then
    {clause.solved[] := true}.

procedure Problem_next()
  self.Problem_nextIter(self.atoms[]).

procedure Problem_nextIter(i)
  self.atom[i].value[] :=
                  self.atom[i].value[].not();
  if self.atom[i].value[] == false then
    {self.Problem_nextIterHelp(i)}.

procedure Problem_nextIterHelp(d)
  if d == 1 then
    {self.finished[] := true};
  if d <> 1 then
    {self.Problem_nextIter(d.sub(1))}.
```

```
import Boolean.
import Number.

procedure start_Problem_solve()
  new A; new B;
  new C1; new C2; new C3;
  new L1; new L2; new L3;
  new L4; new L5; new L6;
  system.atom[1] := A;
  system.atom[2] := B;
  system.atoms[] := 2;
  system.clause[1] := C1;
  system.clause[2] := C2;
  system.clause[3] := C3;
  system.clauses[] := 3;
  C1.literal[1] := L1;
  C1.literal[2] := L2;
  C2.literal[1] := L3;
  C2.literal[2] := L4;
  C3.literal[1] := L5;
  C3.literal[2] := L6;
  L1.atom[] := A;
  ? L1.negated[];
  L2.atom[] := B;
  ? L2.negated[];
  L3.atom[] := A;
  ? L3.negated[];
  L4.atom[] := B;
  ? L4.negated[];
  L5.atom[] := A;
  ? L5.negated[];
  L6.atom[] := B;
  ? L6.negated[];
  C1.try[] := 1;
  C2.try[] := 1;
  C3.try[] := 1;
  C1.pair[] := C2;
  C2.pair[] := C3;
  C3.pair[] := C1;
  system.Problem_solve().

procedure Problem_solve()
  self.finished[] := false;
  while self.finished[] == false do
    {self.Problem_eval()}.

procedure Problem_eval()
  self.solved[] := true;
  foreach c in self.clause do
    {if self.clause[c].try[] ==
              self.clause[c].pair[].try[] then
      {self.clause[c].Clause_try(
                      self.clause[c].pair[])}};
  if self.solved[] == true then
    {self.finished[] := true};
  self.Problem_next().

procedure Clause_try(pair)
  if self.literal[self.try[]].negated[] <>
pair.literal[pair.try[]].negated[] then
    {system.solved[] := false}.

procedure Problem_next()
  self.Problem_nextIter(self.clauses[]).

procedure Problem_nextIter(i)
  self.clause[i].try[] :=
self.clause[i].try[].add(1);
  if self.clause[i].try[] ==
                      self.atoms[].add(1) then
    {self.Problem_nextIterHelp(i)}.

procedure Problem_nextIterHelp(d)
  self.clause[d].try[] := 1;
  if d <> 1 then
    {self.Problem_nextIter(d.sub(1))};
  if d == 1 then
    {self.finished[] := true}.
```

Figure 13. Left column: MOL program generated from Version 1; right column: same for Version 2

As it turns out, iMOL's equivalence check shows that Version 1 and Version 2 of the model are not equivalent. Closer inspection reveals that this is so because Version 1 delivers an actual solution in

terms of truth values (the objects `true` and `false`) assigned (or linked) to the atoms `A` and `B`, whereas Version 2 leaves the initial object structure unaltered, i.e., truth values unassigned. This difference is reflected in the final states the corresponding automata reach: as detailed in Section 5, the states correspond directly to variable assignments in MOL and to object structures (object diagrams) in UML-A. Fortunately, the difference in observable behaviour of the two models, which is a side effect of the underlying algorithms, is easily removed by explicitly unassigning variable values in Version 1 as the final step before model termination (a kind of finalization), and by cleaning up the helper attributes of Version 2 accordingly. The code that needs to be added is shown in Figure 14; after adding it, the models are indeed proven equivalent.

```
procedure start_Problem_solve(user)
    …
    self.Problem_solve();
    A.value[] :-;
    B.value[] :-.

procedure Clause_eval(self)
    …
    self.solved[] :-.
```

```
procedure start_Problem_solve(user)
    …
    self.Problem_solve();
    C1.pair[] :-;
    C2.pair[] :-;
    C3.pair[] :-;
    C1.try[] :-;
    C2.try[] :-;
    C3.try[] :-.
```

Figure 14. Code unassigning helper attributes, needed to make models fully equivalent. Left: Version 1; right: version 2.

A final note on time and space efficiency: the iMOL automaton generation takes less than 4 minutes and approx. 350 MByte of main memory (half of which is needed for the presentation of the resultant automaton) on a PC with Windows 2000 and Pentium III mobile processor (1133 MHz) for Version 1 (with 9437 states), and approx. 2.5 minutes (270 MByte) for Version 2 (7206 states). The equivalence check takes approximately four hours and needs 240 MByte of RAM, most of which is required by the internal representation of the checked automata. Profiling shows that much time is spent in garbage collection; so far, iMOL is a pure research prototype and has not been optimized for size or speed.

8 Discussion

8.1 Applicability

Quite clearly, the definition of UML-A and its translation to MOL make sense only if executability of a model is desired. If on the other hand UML is to be employed as a pure specification language, without at the same time intending to provide a model of an implementation meeting the specification, executability may appear unneeded, and all limitations made to achieve it wasted. Such would typically be the case when specifying interfaces, for example for frameworks or component libraries. However, executability can still be handy in these cases, for instance to validate a specification, or to check compliance with it.

8.2 Language design

From a programmer's perspective, some of MOL's properties may appear debatable. For instance, the absence of `null` or `nil` as a special object (indicating the undefinedness of a variable or attribute) entails that a program may behave arbitrarily (including immediate termination in an error state) once an undefined value is encountered. On the other hand, this frees the modeller from checking for undefinedness of values, a task that would spoil the structure of every model. Since the theoretical properties of MOL allow the automated proof that any variable (or attribute) has a value before its being used, this feature of MOL should be seen an advantage rather than a deficiency.

Another peculiarity of MOL is that it has no dynamic binding. This may appear unusual for a language that is designed for object-oriented modelling, but is a tribute to the fact that dynamically bound procedure calls make programs difficult to trace, counteracting the idea of model validation through execution. Besides, UML's dispatch policy is undefined [11], with most users assuming the (usually single) dispatch mechanism of their favourite programming language. A consensus on this topic is certainly desirable.

One true deficiency of our definition of MOL as the target language for object-oriented model execution is its lack of an asynchronous communication mechanism and of active objects. Although the introduction of messages as first class objects, a message passing mechanism, and parallel processes would be an obvious remedy, we would not want to spoil modelling with the necessary synchronization issues. Besides, it is still unclear to us how parallelism would affect the expressiveness of the formalism; more work needs to be invested here.

8.3 OCL and Action Semantics

UML as a graphical language suffers from a certain clumsiness when it comes to expressing less than trivial things. The designers of the language are well aware of this circumstance and have therefore complemented the graphical notation with a language for expressing constraints (the object constraint language OCL) and a framework for specifying behaviour through atomic actions (the so-called action semantics[5]). Both extensions caricature the idea of graphical modelling to the extent that they require translation of the parts of the diagrams they apply to into a textual form before they can be used to add information inexpressible in the graphical part of UML. For instance, in order to be able to navigate an association, this association must be mapped to an attribute (the dot notation) first. Users of these languages must therefore be able to translate UML diagrams to some linearized form (or at least be able to read this form), which begs the question why it was not used in the first place.

[5] Note that the term *action semantics* is originally occupied by the formal languages community [27], where it has a different meaning (and one not useful for the discussion in this context at that).

Be it as it may, the addition of these two formalisms extends the expressiveness of UML at least to that of the more powerful of the two; any restriction made in our prior reasoning, especially as concerns finiteness of the state space of a model, can therefore—in principle at least—be circumvented. However, since we expect OCL to be used to add constraints to an otherwise underconstrained model, a model with OCL rules will not have more states than the same without; hence, if a model is finite, adding OCL rules cannot make it infinite. Although we might consider adding (excerpts of) OCL to our framework at a later stage, we can safely ignore it for the discussion of our current work.

Action semantics has been introduced to UML only in its last version before 2.0, version 1.5 [30]. It comprises primitive operations for the creation and destruction of objects and links between them, assignment of values (objects) to attributes and variables, iterators for the convenient dereferencing of many-valued attributes and to-many associations, operations for the issuing of signals etc. In particular, as opposed to MOL's being navigational, UML's action semantics is explicitly committed to the relational data model.

Action semantics actions are grouped into procedures which can be tied to the states of a statechart. According to [25], these procedures are executed upon entry of a state. Because a state has no knowledge of the transition that led to it (and its triggering event for that matter), the same procedure is executed independently of which event actually led to the state. Consequently, in the examples shown in [25], all transitions leading to the same state are labelled with the same events, making the statecharts degenerate. As a result, it seems that the statecharts used in UML's action semantics are ordinary flowcharts in disguise (with standardized branching, the branch condition being the occurrence of an event), but without the call of subroutines. To compensate for this deficiency, procedures can contain arbitrary code, so that UML diagrams with action semantics are Turing equivalent. In fact, modelling with action semantics as proposed in [25] more or less means scattering object-oriented code around diagrams, again begging the question why action semantics does not replace for UML completely. With its navigational interpretation of UML (based on attributes rather than associations), MOL captures much of the essence of statecharts with actions, which is nicely reflected in the fact that MOL programs are translated to finite automata, in which states are defined by attribute and variable assignments (Section 5). However, these states lack the abstraction usually associated with statecharts, as can be told by their sheer number (see Section 7).

8.4 Model Checking

Model checking is widely used today to automatically verify properties of finite state systems (e.g., switching circuits or communication protocols) [9]. The properties are specified mostly in linear temporal logic (LTL), and systems are modeled by deterministic finite automata (or Büchi automata). Al-

gorithms developed in formal language theory are then used to determine efficiently whether the system satisfies its given specification.

This idea has also been pursued in the context of UML (e.g., [6, 37]). The popular model checker SPIN [22] is used in all these approaches, in which UML diagrams are translated into PROMELA programs (PROMELA being the system description language used by SPIN). The model checker is then invoked to check deadlock freeness, test system invariants, and check temporal claims.

Typical properties that can be specified by LTL formulas and verified by model checkers are of the form "something good always happens" or "something bad never happens" [6]. In formal language theory terms, LTL is of the same expressive power as first-order logic with successor, and both logics can specify exactly the aperiodic (or, star-free) languages, a subset of the recognizable (regular) languages. In particular, what cannot be formulated in LTL is equivalence of systems, where two systems are said to be equivalent if for all input sequences they produce the same outputs (*cf.* Section 5.3).

Our modelling object language MOL on the other hand was designed so that certain computational problems including the equivalence problem (as well as all properties definable in LTL) become efficiently decidable. In fact, we have implemented an efficient algorithm that can decide in time proportional to $O(N^2)$ whether two given MOL programs (as compiled from UML diagrams) are equivalent in the sense that the same communication with a user (i.e., an actor) produces in both models the same object structure, that is, the same sets of existing objects with the same interrelations between objects [1]. This property (of equivalence of models) is not definable by an LTL-formula, hence we gain decidability over systems based on PROMELA/SPIN. On the other hand, we have not (yet) implemented a check of arbitrary properties formulated by the user in terms of LTL-formulas; hence SPIN is more flexible than our system here.

8.5 Other related work

It is common practice to tie the specification of semantics to the UML metamodel (e.g., [14, 12, 33]). This has the advantage that the UML specification remains self-contained, in particular, that its syntax and semantics are specified in one place. On the other hand, the self-containment of the UML definition makes it circular, and every change of the UML concrete syntax entails a rewrite of the definition, because the metalanguage changes with it [39]. Besides, practical attempts to use the UML metamodel for semantics specification suffer from its unwieldiness and often require adaptations that are not easy to establish. For instance, in [12] it is suggested to extend the UML meta model into a model interpreter by adding so-called meta operations. Operational semantics are specified by means of collaboration diagrams formalized as graph transformation rules, mapping the state (coded as an object graph) before an operation to the state after the operation. The expressive power of the formalism appears not to be limited; results comparable to ours have not been discussed.

The modelling language ALLOY is a small ("minimal") language allowing the description and manipulation of object structures [23]. ALLOY is based on Z; it is strongly typed and offers additional constraints to further restrict the possible state space. ALLOY comes with strong tool support, allowing the explorative development of models and evaluation of language design. However, ALLOY is not designed to model object interactions, which we deem fundamental to object-oriented modelling.

With his abstract state machines (ASMs) Gurevich has extended finite automata by a notion of state that is bound to variable assignments [19]. ASMs are Turing equivalent and can be employed for program specification, supposedly on arbitrary levels of abstraction. Even though the variable assignments of a state roughly correspond to the state of an object as determined by the links it has to other objects, it appears that ASMs are only sporadically used to integrate statecharts with the rest of UML.

ASMs have been extended into the textual modelling language ASML. ASML has many properties of procedural programming languages; its primary innovation appears to be that it allows the combination of statements into atomic steps marking a transition from one state to the next. Because of its programming language syntax and the lack of a suitable graphical notation, however, ASML specifications will be unintelligible to most customers, making them unsuitable for validation purposes.

Ober has formalized large parts of UML as ASMs [29], but despite the obvious formal relatedness has excluded statecharts (because of their enormous complexity; personal communication). Börger et al. separately mapped UML statecharts [2] and activity diagrams [3] to ASMs, avoiding the diagram integration problem. More recently, Cavarra et al. have provided a mapping of UML static structure diagrams to initial ASM signatures [7] and developed a simulator for UML models based upon this mapping [8].

As mentioned above, ASMs in general are computationally as powerful as unrestricted Turing machines; hence the equivalence problem or reachability questions such as the halting problem are undecidable in this model. ASMs are therefore not suitable for our purpose: we want to exploit a lack of expressiveness in graphical modelling language to gain decidability of these questions.

With their MINERVA system Campbell et al. created a visual round-trip tool allowing the validation of class and state diagrams by means of a model checker [6]. Although their system appears to include animated visualization of collaborations, it targets at generating proofs primarily for embedded systems. Embedded systems are also the target of HUGO [37], a system for checking the consistency of dynamic specifications in the form of UML statecharts and collaboration diagrams. It uses PROMELA and the model checker SPIN as its target formalisms (*cf.* above). However, model checking does not lend itself to model validation through animation (i.e., execution and observing its behaviour); we suspect that the constraints that need to be proven cannot be formulated by the majority of UML users.

9 Conclusion

We have presented a framework for an executable and ultimately also validatable subset of UML by providing operational semantics that maintains some of the practical restrictions graphical object-oriented models possess when compared to object-oriented programs. Within this framework we can simulate modelled object interactions and the structural changes they imply, and at the same time prove certain interesting properties of the model, e.g., whether or not an interaction comes to an end, and whether two models are equivalent (or else what the differences between these models are). Although much work needs to be done, we believe that our approach is a worthwhile contribution towards model-driven architecting of software systems.

References

1. Baselau, S.: Äquivalenz von UML-Diagrammen. Diplomarbeit (Universität Hannover, 2004).
2. Börger, E., Cavarra, A., Riccobene, E.: Modeling the dynamics of UML state machines. In: Gurevich Y., Kutter P.W., Odersky M., Thiele L. (eds.) Abstract State Machines. Springer LNCS 1912 (2000) 223–241.
3. Börger, E., Cavarra, A., Riccobene, E.: An ASM Semantics for UML Activity Diagrams. In: Rus, T. (ed.): Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000 (2000) 293–308.
4. Breslauer, D.: The suffix tree of a tree and minimizing sequential transducers. Theoretical Computer Science, 191:1-2 (1998)131–144.
5. Buchloh, T.: Entwurf und Implementierung eines Interpretierers für die Sprache MOL. Bachelor-Arbeit (Universität Hannover, 2003).
6. Campbell, L. A., Cheng, B. H. C., McUmber, W. W., Stirewalt, R. E. K.: Automatically detecting and visualizing errors in UML diagrams. Requirements Engineering 7:4 (2002) 264–287.
7. Cavarra, A., Riccobene, E., Scandurra, P.: Integrating UML Static and Dynamic Views and Formalizing the Interaction Mechanism of UML State Machines. In: Börger, E., Gargantini, A., Riccobene, E. (eds.): Abstract State Machines, Advances in Theory and Practice, 10th International Workshop (2003) 229–243.
8. Cavarra, A., Riccobene, E., Scandurra, P.: A framework to simulate UML models: moving from a semi-formal to a formal environment. In: Proceedings of the 2004 ACM symposium on Applied computing SAC 2004 (2004) 1519–1523.
9. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems 16:5 (1994) 1512–1542.
10. de Champeaux, D.: Letter to the editor. Communications of the ACM 46:3 (2003) 11–12.
11. Edwards, J., Jackson, D., Torlak, E.: A type system for object models. In: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering (2004) 189–199.

12. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: a Graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans A., Kent S., Selic B. (eds.): UML 2000 Proceedings of the 3rd International Conference Springer LNCS 1939 (2000) 323–337.
13. Engels, G., Huecking, R., Sauer, S., Wagner, A.: UML Collaboration Diagrams and their Transformation to Java. In: France R., Rumpe B. (eds.): UML 99 Proceedings of the 2nd International Conference. Springer LNCS 1723 (1999) 473–488.
14. Evans, A.S., Kent, S.: Core meta-modelling semantics of UML: the pUML approach. In: France R., Rumpe B. (eds.): UML 99 Proceedings of the 2nd International Conference. Springer LNCS 1723 (1999) 140–155.
15. Even, S.: Graph Algorithms. Computer Science Press (1979).
16. Fairley, R.E. Software engineering concepts. McGraw-Hill (1985).
17. Gamma, E., Helm, R., Johnson, R. E., Vlissides, J.: Design Patterns. Addison-Wesley (1995).
18. Gill, A. Introduction to the Theory of Finite-State Machines. McGraw-Hill (1962).
19. Gurevich, Y. Sequential abstract-state machines capture sequential algorithms. ACM Transactions on Computational Logic 1:1 (2000) 77–111.
20. Hagemann, P. UML-Diagramme und ihre Übersetzung in die Modell-Objekt-Sprache MOL. Diplomarbeit (Universität Hannover, 2003).
21. Hopcroft, J. E. , Motwani, R. Ullman, J. D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Boston, 2nd edition (2001).
22. Holzmann, G. J.: The model checker SPIN. IEEE Transactions on Software Engineering 23:5 (1997) 279–295.
23. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology 11:2 (2002) 256–290.
24. Kfoury, A. J. , Moll, R. N., Arbib, M. A.: A Programming Approach to Computability. Texts and Monographs in Computer Science. Springer-Verlag, New York Heidelberg Berlin (1982).
25. Mellor, S. J., Balcer, M. J.: Executable UML. Addison Wesley (2002).
26. Mohri, M.: Minimization algorithms for sequential transducers. Theoretical Computer Science, 234 (2000) 177–201.
27. Mosses, P. D.: Action Semantics. Cambridge University Press (1992).
28. http://research.microsoft.com/foundations/asml/
29. Ober, I.: Harmonizing Design Languages with Object-Oriented Extensions and an Executable Semantics. Dissertation (Institut National Polytechnique de Toulouse, 2001).
30. OMG Unified Modeling Language Specification Version 1.5 (www.omg.org, 2002).
31. Palsberg, J., Schwartzbach M. I.: Object-oriented type systems (Wiley, 1994).
32. www.puml.org
33. Riehle D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N. The architecture of a UML virtual machine. In: Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (2001) 327–341.
34. Schöning, U.: Theoretische Informatik – kurz gefasst. Hochschultaschenbuch. Spektrum Akademischer Verlag, Heidelberg Berlin, 4th edition (2001).

35. Sénizergues, G.: L(A)=L(B)? Decidability results from complete formal systems. Theoretical Computer Science 251 (2001) 1–166.
36. Sénizergues, G.: T(A)=T(B)? In: Wiedermann J., Emde Boas P.v., Nielsen M. (eds.): Proceedings 26th International Colloqium on Automata, Languages and Programming, Springer LNCS 1644 (1999) 667–676.
37. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. Electronic Notes in Theoretical Computer Science 55:3 (2001).
38. Stapel, K.: Transformation objektorientierter Programmtraces in Animationsskripte eines 3D-Renderers. Bachelor-Arbeit (Universität Hannover, 2003).
39. Steimann, F.: A radical revision of UML's role concept. In: Evans, E., Kent, S., Selic, B. (eds.): UML 2000: Proceedings of the 3rd International Conference. Springer LNCS 1939 (2000) 194–209.
40. Steimann, F., Gößner, J., Mück, T.: On the key role of composition in object-oriented modelling. In: Stevens, P., Whittle, J., Booch, G. (eds.): UML 2003: Proceedings of the 6th International Conference Springer LNCS 2863 (2003) 106–120.
41. Steimann, F., Kühne, T.: A radical reduction of UML's core semantics. In: Jézéquel, J. M., Hussmann, H., Cook, S. (eds.): UML 2002: Proceedings of the 5th International Conference Springer LNCS 2460 (2002) 34–48.
42. Steimann, F., Thaden, U., Siberski, W., Nejdl, W.: Animiertes UML als Medium für die Didaktik der objektorientierten Programmierung. In: M Glinz, Müller-Luschnat (eds.): Modellierung 2002 GI Lecture Notes in Informatics P-12 (2002) 159–170.
43. Stirling, C.: Deciding DPDA equivalence is primitive recursive. In: Widmayer P., Triguero F., Morales R., Hennessy M., Eidenbenz S., Conejo R. (eds.): Proceedings 29th Colloqium on Automata, Languages and Programming, Springer LNCS 2380 (2002) 821–832.
44. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J. M.: Refactoring UML Models. In: Gogolla M., Kobryn C. (eds.): "UML" 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools Springer LNCS 2185 (2001) 134–148.
45. Wegener, I.: Theoretische Informatik – eine algorithmenorientierte Einführung. Leitfäden der Informatik. Teubner, Stuttgart (1999).