A Radical Reduction of UML's Core Semantics[*]

Friedrich Steimann and Thomas Kühne

Institut für Technische Informatik
Rechnergestützte Wissensverarbeitung
Universität Hannover, Appelstraße 4
D-30167 Hannover
steimann@acm.org

Praktische Informatik
Technische Universität Darmstadt
Wilhelminenstraße 7
D-64283 Darmstadt
kuehne@pi.informatik.tu-darmstadt.de

Abstract. UML's current core semantics suffers both from excessive complexity and from being overly general. Resultant is a language definition that is difficult to master and to repair. This is the more disturbing as the current core and its extensions do very little to integrate statics and dynamics, even though the inseparability of these is a property of software from which many of the modelling difficulties arise. To better this unsatisfactory situation, we suggest a simple modelling core with few concepts that are easy to understand, yet cover most static and dynamic modelling aspects. We present our work, which is founded in elementary set theory, in natural language making it equally accessible for both practitioners and formalists.

1 Introduction

Much has been written about the shortcomings in the current UML specification, which almost paradoxically contrasts the standard's considerable length. Without adding to the ever expanding list of inaccuracies and inconsistencies, we maintain here that many of the standard's flaws are rooted in its foundation, particularly in its core package and associated semantics. However, the same unwieldiness that left the faults undiscovered by its designers makes it difficult to evolve the standard into something sound. Therefore, we propose a radically revised language core that builds on a small set of primitive modelling concepts, with new abstract syntax and formal semantics. We demonstrate that this new core is to a large extent compatible with UML's current concrete syntax and pragmatics.

One of the key points of our contribution is that the language we propose is designed right from the bottom up with a strong emphasis on integrating statics and dynamics, so that the same small core can serve as the basis for both the static and the dynamic views on a system. So far, a formulation of such an underlying model has been hinted at only, leaving the various views offered by the language largely disconnected. We are well aware, though, that much remains to be done to cover the whole of current UML syntax and semantics. Therefore, our core language can only be a language core, to be extended wherever deemed necessary.

The remainder of this paper is organized as follows. We start by briefly reviewing our choice of core modelling concepts and then define a minimal abstract syntax

[*] in: JM Jézéquel, H Hussmann, S Cook *UML 2002: Proceedings of the 5th International Conference* (Springer, 2002) 34–48.

which is based on a blend of order-sorted logic and set theory. The formal semantics of this syntax is heritage from corresponding formal frameworks offered by the logic programming community [1] and hence only touched on. Following, we show the mapping of UML diagram types to this abstract syntax. A brief discussion concludes our paper.

Before proceeding with the presentation of the basic concepts on which we build our language definition, we would like to stress one subtlety that often leads to confusion. Generally, we do not speak of a core model, but of a core (modelling) language, since we do not intend to *model* UML (just like a UML user would model any other subject domain). Rather, we rely on elementary set theory (our metalanguage) to specify a language (our object language) the sentences of which are interpreted as models of software systems. The semantics of standard UML diagrams is defined by providing a set of mapping rules to our object language, which, being based on set theory, has standard semantics. By not following the common metamodelling approach favoured by the OMG we do not only avoid circular definitions, but also the unfortunate situation that every change of the UML concrete syntax entails rewriting of the whole definition, because the metalanguage changes with it.

2 Core modelling concepts

2.1 Time

Conceptual modelling and its most prominent semantic foundation, formal logic, have a long tradition in ignoring time: even though change and hence time are the very nature of software systems, the ER(= basically predicate logic)-based strain of modelling tends to neglect the ubiquitous duality of static and dynamic viewpoints. For instance, multiplicities (or cardinalities) apply to the contemporaneous number of links between objects (or entities) and hence place constraints on change rather than static structure (see section on extensions below).

As a consequence we maintain that statics and dynamics are not as orthogonal as many modelling language specifications (including UML) would like them to appear. We try to do away with this inadequacy by putting time at the core of our modelling language. In particular, we regard time as an index to all modelling expressions at the instance level, albeit mostly only indirectly, through state.

Actually, time indices comes in two guises: quantitatively, as absolute or relative points on a linear time scale, and qualitatively, with sequence as the underlying time model. The difference between these two is much bigger than it may seem: specifications of the latter form usually cover a whole lot of alternative evolutions simultaneously (a non-linear or branching time model), whereas specifications based on the former are typically non-branching, showing only a single evolution (usually out of many possible).

2.2 Intension and extension

Intension and extension are properties of concepts. In our setting, the intension of a modelling concept may be equated with its definition or specification. Intensions are usually only changed during the development process of a software system, not while the software is executing. Hence, we regard intension as a static (i.e., atemporal) concept. The reflective capabilities of some OOPLs make an exception to this common assumption, but are not considered here.

The extension of a modelling concept is the set of elements that fall under that concept, i.e., that are adequately described by the concept's intension. Unlike intensions, extensions are inherently dynamic: elements come and leave extensions over time, making them grow and shrink. The static extension of a concept, if at all a useful notion, may be considered the temporal projection of (or union over all) its dynamic extensions.[1] It must be understood, then, that certain constraints (e.g. cardinality) expressed for the extension of a concept hold only for its dynamic extensions.

2.3 Types

Types are at the heart of modelling. Equipped with intensions and extensions, they serve as specifiers for the properties of sets of objects that can be treated alike, and as constraints expressing objects of which kinds may take which places.

In our core language we need two kinds of types: natural types, which we call *classes*, and role types, which we call *roles*[2]. For readers unfamiliar with the role-as-type concept, roles as used here may safely be equated with the interfaces of UML and JAVA [6]. UML makes a further difference between object and primitive types (misnamed data types); however, this distinction is of no concern in this discourse and hence ignored.

There exists a rather crisp ontological distinction between classes and roles, based on the dynamic extensions of types and associations. Briefly, a type is a class if for an object to belong to that type no engagement in a relationship to other objects is necessary, and changing this type (so-called object migration) means that the object loses its identity. For instance, a person is a person qua being, not per relatedness to any other object, and it cannot stop being a person without losing its identity. A role, on the other hand, is only adopted in the context of an association, and taking on a role or giving it up does not change the object's identity. For instance, a father is only a father if he has children, but becoming a father does not change the individual's identity. As an aside, note that unlike classes, roles have no instances of their own; they must recruit them from natural types, i.e., classes.

The concept of roles in UML has been thoroughly revised in [4]; its equality to interfaces has been argued for elsewhere [6, 7]. Readers feeling uncomfortable with the prominent role roles play in this treatise may continue by ignoring roles and the associated modelling issues, albeit only at the price of losing some of the coverage of the more advanced UML semantics (interface specifiers, collaboration roles etc.).

2.4 Objects

The extensions of types are sets of objects. The sets are dynamic, reflecting the fact that objects come into life (technically referred to as instance creation) and disappear (referred to as destruction). Whereas for classes instance creation and destruction

[1] We speak of the dynamic extensions of a concept in plural here since our underlying view is that a concept has a (dynamic) extension at each point in time.

[2] We have taken special care that there is no danger of confusing the type and the instance level when speaking of roles.

alone change the extensions, things are different for roles: the dynamic extension of a role at time *t* is defined as the set of objects occurring — at the same time *t* — in the dynamic extents of any associations, in the place of that role. Thus, the dynamic extensions of roles depend on those of classes and those of associations.

As is commonly understood, in our approach too all objects have identity. Some objects have nothing but their identity, for instance numbers. Objects can be named; in this case, the name serves for referencing the object and may be considered an alias for the identity.

2.5 Type hierarchies and inheritance

Types are organized in hierarchies, one for each kind of type: a *class hierarchy* and a *role hierarchy*. Both are subsumption hierarchies, i.e., each supertype subsumes its subtypes. The connotation of type subsumption here is that the extensions of supertypes include the extensions of all their subtypes. Conversely, the intension of a subtype implies the intensions of all its supertypes, meaning that all properties specified for a supertype also hold for its subtypes. Thus, type subsumption establishes type compatibility; it may be considered equivalent to the *generalization* relationship of UML and entails what is commonly called inheritance in OOPLs.

The class and the role hierarchy are connected through a special relationship extending the type compatibility, by specifying the objects of which classes can appear where which roles are specified. For a quick understanding, this relationship can be equated with the *realizes* dependency of UML and the *implements* keyword of the JAVA programming language, specifying the instances of which classes can be assigned to variables of which interface type. The full semantics of this relationship, especially with regard to intensions and extensions, is a little more complicated and detailed in [3, 5].

2.6 Attributes

Each class comes with a set of attributes. Each attribute is modelled as an unary function from the extension of the class to some value domain, namely the extension of another class. Again, we refrain from making a distinction between object and primitive types. Many-valued attributes (i.e., attributes that can have several values at the same time) are modelled as attributes with sets[3] as the elements of their value type, corresponding to the container types of OOP.

There is a certain degree of freedom as to whether or not a relationship between two objects should be modelled as an attribute or as a structural association (see below). Although there are some theoretical hints on how to decide this question, the issue is of no importance for the further discourse and can safely be ignored.[4] We stress here, though, that our focus is on the conceptual side of modelling, not on implementational issues.

[3] Mathematically, tuples, sequences and bags (multisets) can all be modelled as sets.

[4] Just a quick ontological hint: attributes usually model qualities of an object, not relationships; hence classes are preferred over roles as value types.

2.7 Associations

Objects can engage in relationships with certain other objects. This property is specified on the type level, by declaring the involved types as arguments to associations. The declaration is part of the association's intension. Other properties usually complement the declaration, but are not considered here.

Associations are generally interpreted as relations in the mathematical sense, i.e., as subsets of the Cartesian products of the extensions of the involved types. The dynamic extensions of an association are tuples called links (Section 2.8 below). Note that this definition leaves it open how associations are to be used, i.e., whether they define structure or collaboration (see [7] for a deeper discussion). Because this is an important distinction, we subdivide associations into structural and procedural associations below.

It is useful to require that associations are defined on roles only. It has been shown in [4] that this elegantly covers many of the lesser used model elements of UML's static structure and collaboration diagrams without posing unnecessary restrictions on current UML pragmatics. However, to maintain the independent readability of this paper we continue without insisting on this separation.

Structural associations. Structural associations model the knowledge an object may have of its environment. In particular, they represent the possible relationships between objects that exist independently of the execution of some procedure.

Procedural associations. A procedural association declares the types of the parameters contributing to a procedure, i.e., to a piece of functionality required for the software system to achieve its purpose. The places of a procedural association are

- one (the first) for the object responsible for executing the procedure (the receiver),
- one for each other parameter of the procedure, and
- optionally one (by convention the last) for the return value or result of the procedure.

Note that in OOP the understanding is that messages are sent to a receiver. In our modelling language, the coupling of a procedural association with its receiver type is rather loose (in fact, it is not tighter than that of a structural association with any of its types).[5] The underlying design decision for the last point is not to introduce return actions as in standard UML, but to view functions (methods that return values) as special relations in the mathematical sense.

Aggregation. The concept of aggregation has been the subject of much debate in both the philosophical and conceptual modelling communities. We argue here that because we have no useful standard semantics for the concept of aggregation that should be captured in the core language, there is no good in introducing it; hence we leave it. A possible realization of aggregation at the modelling level is shown in Figure 2.

[5] This has some theoretical impact on the resolution of dynamic binding (method dispatching), which is not treated here. See [1] for a formal treatise.

2.8 Links

Links are the elements of the dynamic extensions of associations. They are basically tuples of objects, complemented by the name of the association to whose extension they belong. As with associations, we distinguish between structural links and procedural links.

Structural links. Structural links model the relationships between objects that express the knowledge an object has of others, often independent of any particular functional requirement. In particular, structural associations exist before and after the execution of a procedure, but they are often the result of some procedure.

Procedural links. Procedural links express relationships existing solely for the purpose of realizing some functionality or, more technically, for executing some procedure. Procedural links are transient by nature; their creation corresponds to the event of a method call (with objects as actual parameters) and their destruction to the end of execution of the called method. A procedural link may survive the procedure by being transformed into a structural link (see Figure 3 for an example).

2.9 States

The state of an object-oriented software system (the *system state*) is the set of its contemporaneously existing objects together with their attribute value assignments (where some objects are the attribute values of others) and the links (both structural and procedural) between them. Due to practical limitations states will usually only be partially specified, i.e., each given state represents only a small excerpt of some total system state.

Even though it is understood that a state is a snapshot, single, isolated states are usually not linked to absolute times since it is the very nature of a software system that its state at any one time depends on past states and events that have occurred since then. Instead, states occur in the definition of state transitions and state/event sequences (see below).

The state of an object is the set of its attribute value assignments and the links it shares with other objects. It comprises the knowledge an object has of itself and of its environment.

2.10 Events

An event corresponds to the creation or destruction of one or more procedural links. Since procedural links contribute to state, any event partially specifies its own successor state. This makes sense because the information conveyed in a procedural link, namely the objects passed as parameters, should at least be temporarily known to the object responsible for the execution of the procedure.

In the absence of classes as first class objects instance creation (i.e., a change to the extension of a class) cannot be triggered by an event as defined above, but must be an event in its own right. However, unlike the creation of a procedural link, the creation of a structural link is not an event; rather, it is a reaction to an event reflected in a change of the dynamic extension of a structural association.

Events can be timed, meaning that an absolute time or a time relative to the occurrence of another event can be associated with it.

2.11 State transitions

A state transition is a mapping from a state and an event to a successor state. In the context of software modelling, two forms of state transitions are of interest: sets of partial specifications of the transition function of a finite state machine in the form "if in state s_1 and event e occurs, change to state s_2", and the listing of state/event-sequences of the form $<s_1\ e_1\ s_2\ e_2\ ...\ s_n>$. A (complete) transition function specifies a set of possible sequences of events and thus a formal (regular) language; state/event-sequences specify one concrete sequence of events and resultant state changes. Whereas in the former case no absolute time tags can be attached to events or states, such may be useful in the latter case, for instance for specifying a real-time scenario.

Just as a state can be associated with both the whole system and each object, so can state transitions: a state transition defined for an object starts from a state of that object and arrives at a state including the new state of that object, plus possibly the changed states of other objects involved and procedural links newly created. State transitions of this kind are usually associated with the object's type.

3 Core language

With modelling concepts defined as above at hand, we are now equipped to define our core object-oriented modelling language. We do this by first defining the abstract syntax, i.e., by specifying how a model is textually expressed in our language. In Section 4 we show how standard UML diagrams are mapped to this abstract syntax. It must be understood, though, that our core language was not designed to cover everything offered by the UML standard; however, it is only a core that can easily be extended to cover more meaning.

3.1 Abstract syntax

Expressed in our abstract syntax, a UML model consists of

- a finite set **C** of *class symbols*;
- a finite set **R** of *role symbols*;
- a transitive, reflexive and antisymmetric overloaded relation $\leq$ defined on $\mathbf{C}\times\mathbf{C}$, $\mathbf{R}\times\mathbf{R}$, and $\mathbf{C}\times\mathbf{R}$, expressing the *combined type hierarchy of classes and roles*;
- for each $C\in\mathbf{C}$ a set of *attribute declarations* of the form a: $C \rightarrow D$ where a is the name of the attribute and $D\in\mathbf{C}$ the value type;
- a finite set **S** of *structural association declarations* of the form s: $R_1\ ...\ R_n$ where s is the name of the structural association and $R_1, ..., R_n\in\mathbf{R}$ are the types of the association's places;
- a finite set **P** of *procedural association declarations* of the form p: $R_1\ ...\ R_n$ where p is the name of the procedural association and $R_1, ..., R_n\in\mathbf{R}$ are the types of the association's places;
- a finite set $\mathbf{\Sigma}$ of states with each state $\sigma\in\mathbf{\Sigma}$ comprised of
  - a (possibly empty) set O_C of object symbols for each $C\in\mathbf{C}$ (the dynamic extension of C in state σ);

- an optional attribute value assignment of the form $a(o) = u$ for each object $o \in O_C$ and attribute declaration $a\colon E \to F$ with $C \leq E$ and $u \in O_D$ for some $D \leq F$ (inheritance and type or assignment compatibility!);
- a (possibly empty) finite set L_s of links of the form $s(o_1, ..., o_n)$ for each structural association declaration $s\colon R_1 ... R_n \in \mathbf{S}$, where $o_1 \in O_{C_1}$, ..., $o_n \in O_{C_n}$ and $C_1 \leq R_1$, ..., $C_n \leq R_n$ (the dynamic extension of s in state σ);
- a (possibly empty) finite set L_p of links of the form $p(o_1, ..., o_n)$ for each procedural association declaration $p\colon R_1 ... R_n \in \mathbf{P}$, where $o_1 \in O_{C_1}$, ..., $o_n \in O_{Cn}$ and $C_1 \leq R_1$, ..., $C_n \leq R_n$ (the dynamic extension of p in state σ); and

- a finite set of exemplary state transitions of the form $\delta(\sigma_1, l) = \sigma_2$ with $\sigma_1, \sigma_2 \in \mathbf{\Sigma}$ and $l \in L_p$ for some $p \in \mathbf{P}$.

A few comments are in place:

- One might maintain that roles (i.e. interfaces) should appear only at the target ends of unidirectional associations, presumably because interfaces cannot have attributes in JAVA. We do not support this viewpoint.
- As indicated above, the alternative form of declarations $s\colon T_1 ... T_n$ and $p\colon T_1 ... T_n$ with T_1, ..., $T_n \in \mathbf{C} \cup \mathbf{R}$ is also possible, but less expressive; see [4].
- The state transitions of the given form cover the specification of both state machines and a non-branching sequences of state/event pairs.
- Quite obviously, much of the complexity of modelling lies in the specification of states and state transitions, and thus in the dynamic (i.e., time-indexed) parts of a model. All attempts to increase the expressiveness at the (atemporal) type level must appear misled investments by comparison.

The above definition of a core abstract syntax uses natural language and the formal notation of set theory and order-sorted predicate logic [1, 3]; it could also be depicted graphically, for instance in UML, given that a UML diagram can be drawn that has the same meaning as the above prose. Because UML is intended for software modelling and not for defining a language, we have no interest in doing so and since leave it as an academic exercise.

3.2 Semantics and the family of language issue

Since our core modelling language builds on the mathematical primitives set, function, and relation, the definition of its semantics in set theory is straightforward. A short definition for a similar language based on order sorted predicate logic has been given in [3]; lengthier treatments can be found in [1].

It must be noted, though, that certain issues such as overloading and binding (which links are instances of which associations) are not unambiguously defined with the language syntax and must be explicitly stated in its semantics. By offering alternative interpretations the same syntax can serve a whole family of modelling languages. Because of its commitment to types, however, it is illusory to pretend that our language could be equally useful for fundamentally different approaches such as prototype-based modelling.

3.3 Shortcomings

The presented language core strictly distinguishes between types and objects. In particular, whereas objects can have attribute values ("slots") and can be involved in links, classes cannot. This inhibits, among other things, "static" declarations and instance creation through sending a class a message. Two solutions are possible: either we introduce a special type *Type* whose instances act as proxies for the ordinary types, or we introduce a metatype level to our object language and let the non-metatypes be instances of metatypes. The former resembles the type system of JAVA, even though without further measures it can do little more than instance creation[6]. The latter results in a type model similar to that of SMALLTALK; in particular, if all modelling constructs applying to types also apply to metatypes, types may change dynamically, yielding a fully reflective language. Be it as it may, the shortcoming is not as dramatic as it my appear for many programmers, since static features rarely play a role in OOM.

Another shortcoming of our language is that it makes no provisions for capturing technical modelling information such as from which diagram a certain model element originates. This is particularly a problem since our aim is to integrate the information of all diagrams into a single representation, so that some information is necessarily lost. However, separate storage of such information is not a problem technically, so that we leave this subtlety aside.

3.4 Core extension

Since our core language is founded on set theory, possible extensions that are theoretically sound and practically useful abound in the literature. It must be understood, though, that most extensions will not lead to the addition of modelling concepts to our language core, but to some standardized formulation of constraints to be placed on models. For instance, cardinality constraints associated with an association translate to expressions requiring that only a certain number of links exist contemporaneously with the same object involved. It appears that the Object Constraint Language (OCL) [2] would lend itself to expressing such constraints; its relationship to our core modelling language, however, must be the subject of another paper.

4 Mapping of UML diagram types

Now we come to the final hurdle of our venture: how do UML's different diagram types map to the suggested core modelling language? After all, our main intent is to keep UML as a notation intact but redefine what the modelling information expressed by the various diagram types is. This not only gives each diagram type a well defined meaning, but also tightly integrates the diagram types with each other, namely as different views to a common model. Let us see how this works, one by one.

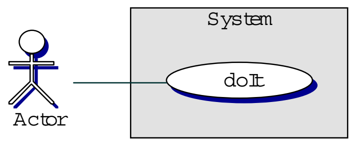
4.1 Use case diagram

A use case maps to a procedural association declaration between an actor and a system. The actor maps to a role symbol, and the system to a class symbol (Figure 1).

[6] because all instances of *Type* must share the same properties

Since associations should end at roles, an implicit interface between the association end and the system class may be assumed, declaring the view of the use case on the system by hiding other properties of the system irrelevant to this use case. An alternative mapping is to view the actor as an interface to the system and the use case as an unary procedural association declared on that interface (= role; see Figure 1). Such a mapping is adequate if the actor is not itself involved in the use case.

Note that uses case, actor and system are all at a very high level of abstraction (and possibly refined as aggregates of procedural associations, roles and classes). The procedural associations of a use case diagram could be instantiated to procedural links with concrete objects at their places, but the use of doing so is rather limited.



classes	classes
System	*System*
roles	roles
Actor	*Actor*
SystemRole	
type hierarchy	type hierarchy
System ≤ *SystemRole*	*System* ≤ *Actor*
procedural associations	procedural associations
doIt: *SystemRole Actor*	*doIt*: *Actor*

Figure 1: A use case and two alternative mappings to our abstract syntax. `doIt` becomes a procedural association, `Actor` a role and `System` a class. *SystemRole* is introduced implicitly.

4.2 Class diagram

The class diagram depicts primarily types and the structural associations between them. Not incidentally, it is also called static structure diagram. However, method (or operation in UML jargon) declarations and thus elements pertaining to behaviour are also shown in a class diagram.

The name of a class maps to a class symbol and that of an interface to a role symbol. Class stereotypes (other than interface and role, which should be mapped to role symbols) are not considered here[7]. Attributes map to attribute declarations from the holding class to the attribute value type. The method declarations of a class are translated to the declarations of procedural associations of which the first parameter is the defining class (Figure 2).[8]

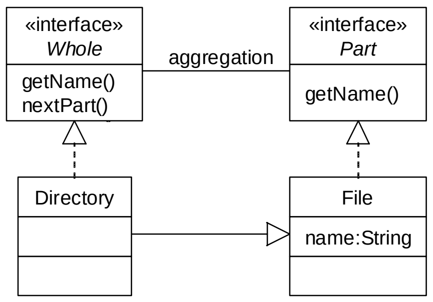
Associations map to structural association declarations (Figure 2). Whether or not an association is an aggregation has no formal implications and is thus ignored here, as are multiplicities (cardinality constraints). UML's restriction disallowing bi-

[7] although they could be addressed by the aforementioned metatype level

[8] Since procedural associations should be defined on roles, not classes, they should really be associated with an interface, namely the interface across which the method is called. However, as we said earlier, in order not to complicate matters, we do not insist on this distinction here.

directional associations between roles (or interfaces) is lifted, basically because to us it seems to lack a plausible justification. If association ends carry rolenames, they are interpreted as implicit introductions of role types, implemented by the classes placed at the association ends.[9]

A generalization arrow between two classes or two interfaces, or an *implements* arrow between a class and an interface is mapped to a pair of the ≤-relation specifying the type hierarchy. If instances and links are shown in a static structure diagram (a mixed class and object diagram), they correspond to object symbols and structural links, respectively. Other static structure diagram elements are not treated here; they may necessitate extensions to the core language.



type hierarchy

File ≤ Part

Directory ≤ Whole

Directory ≤ File

attributes

name: *File → String*

structural associations

aggregation: *Whole Part*

procedural associations

getName: *Whole String*

getName: *Part String*

nextPart: *Whole Part*

Figure 2: A class diagram and its mapping to declarations in our abstract syntax. The diagram models an aggregation pattern (with roles *Whole* and *Part*) and a file system structure as a possible application of it.

4.3 Collaboration diagram

The collaboration diagram marries two different aspects of an interaction: the types of the collaborators together with their associations (or links, depending on whether the diagram is on the specification or instance level), and the message flow indicating the interaction between objects.

The former aspect explicates the links necessary for the communication between objects and could also be shown in an object or class diagram (with one exception listed below), while the latter aspect is also shown in the sequence diagram (see Section 4.4).

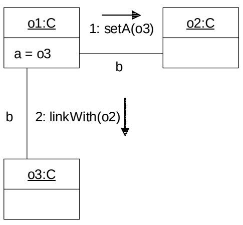
The mapping of collaboration diagrams to the abstract syntax is straightforward. For the instance level, objects in roles map to objects (possibly anonymous, acting as placeholders and corresponding to variables with the roles as their types), links map to structural links and messages to procedural links (with the receiver as the first ar-

[9] For a more thorough treatment of association ends, roles and association generalization, the reader is referred to [4, 5].

gument). The sequence specified is a linear one and mapped to a state/event sequence, even though not all state changes (only object and procedural link creation and destruction and no structural changes) are actually specified. Figure 3 gives an example of this.

For collaboration diagrams at the specification level, classifier roles map to role symbols, classes, if specified, to class symbols (if both are provided for a single classifier, it is understood that the class is subsumed by the role), associations to structural association declarations, and sent messages to procedural association declarations. The message sequence specified maps to a set of state transitions partially specifying a finite state machine. This finite state machine is the rough equivalent of the regular expression specified by the sequence of control structures (i.e., conditionals, repeats, and calls) expressed in the collaboration diagram[10].

(a)



(b)

$$\sigma_1 = \begin{bmatrix} a(o_1) = o_3 \\ b(o_1, o_2) \\ b(o_1, o_3) \end{bmatrix}; \quad l_1 = [setA(o_2, o_3)]$$

$$\sigma_2 = \begin{bmatrix} a(o_1) = o_3 \\ b(o_1, o_2) \\ b(o_1, o_3) \\ a(o_2) = o_3 \end{bmatrix}; \quad l_2 = [linkWith(o_3, o_2)]$$

$$\sigma_2 = \begin{bmatrix} a(o_1) = o_3 \\ b(o_1, o_2) \\ b(o_2, o_3) \\ a(o_2) = o_3 \end{bmatrix}$$

(c)

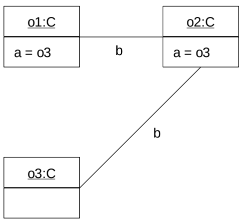


Figure 3: Collaboration diagram and its mapping.
(a) shows a collaboration diagram at the instance level. Objects, attribute values and links show the initial state. (b) shows the same expressed in the abstract syntax, with subsequent states as would be expected (but not shown by the diagram). (c) shows the final state as an object diagram.

Now where does the structural aspect of a collaboration diagram go beyond that of a static structure diagram? With the procedural link the receiving object gains temporary knowledge of the objects involved in the link[11]. This knowledge can be used in several ways:

[10] and which corresponds to the static program text (algorithm) of the procedure
[11] including itself of which, of course, it already has knowledge

- temporarily to construct a new procedural link involving one or more of the received objects and send it to another object known by (or linked to) the receiver;
- temporarily as a structural link for sending one of the received objects a new procedural link; and
- permanently as a new (or as a replacement for an existing) structural link to a received object.

Thus, the existence of a structural association between two types is not a necessary prerequisite for the exchange of a procedural link (a method call) between two objects of these types; instead, a procedural association suffices. Thus, the structural aspects of a collaboration diagram go beyond what is shown in a static structure diagram — which can also include roles — in that it shows the temporary links resulting from passed parameters.[12] It does not, however, show the actual creation of these links as state changes.

4.4 Sequence diagram

A sequence diagram shows objects exchanging messages to achieve a certain purpose. Its primary use is the specification of the sequence of these messages, plus which objects receive them. Because the sequencing is not dependent on a linear numbering of the calls, sequence diagrams are particularly well-suited to express branching by providing state transitions with identical starting states. As with the collaboration diagram, the only state changes that are shown correspond to the creation and destruction of objects, and those of procedural links.

The mapping of a sequence diagram is analogous to that of a collaboration diagram. The actor maps to a role symbol, all objects map to object symbols with their properties specified by the corresponding types. A method call corresponds to the creation of a procedural link and thus to the occurrence of an event. Synchronicity of the call places conditions on the state transitions associated with the caller, namely whether or not it waits for the receiver to have completed the state transitions triggered by the call. Timing constraints expressed with the call sequence are either mapped to timed state/event-sequences (linear time model) or to transition times associated with the state transitions.

4.5 Statechart diagram

Statecharts are extensions of finite state machines and thus, basically, a more expressive means of specifying state transition functions. The mapping of a statechart to a finite state machine must reverse the extension, which is not an easy undertaking.

Even though the specification of a statechart has the directest mapping to the state transitions of our core language in principle, the states of a statechart are usually abstractions of states as expressed in our abstract syntax, which are basically characterized by attribute values and links of objects. Thus, the mapping of the states of a statechart to the core is not at all clear. A possible workaround would be to implicitly specify state attributes with one value for each possible state of an object; this, however, undermines our goal to have all diagram types mapped to a single core model,

[12] In a method implementation these would be the actual parameters plus temporary (i.e. method local) variables.

since it leaves the state transitions specified by interaction diagrams and those specified by statecharts largely uncorrelated.

Generally, finite state machines tend to become very complex even for fairly moderate modelling problems, which is the reason why statecharts have been introduced in the first place. Although expressive enough in principle, it may be worth considering to replace the definition of events, states and state transitions in our core language by (a subset of) the definition of statecharts. This, however, cannot be dealt with here.

4.6 Activity diagram

Activity diagrams are said to be based on state charts, a statement which should facilitate mapping. However, both authors have surrendered to the unwieldiness of the activity diagram specification and abandoned all attempts to provide an even near plausible mapping.

4.7 Component and deployment diagram

Component and deployment diagrams show aspects important for the technical design of a software product. Since their relation to (conceptual) modelling is rather weak, they are not treated here. It is a common observation, however, that structuring mechanisms are often orthogonal to the (elements) of the language they structure; i.e., it is very likely that none of the modelling concepts presented in section 2 will be suitable to serve for any of the purposes pursued by the component and the deployment diagram.

5 Discussion

The UML standard's current (version 1.4) chapter 2 comes with a wealth of additional modelling concepts (even in the core package) which we have not addressed here. However, it is our conviction that the primary goals of a core modelling language should be conciseness (ideally: minimality of concepts) and the ability to serve as a uniform basis for all modelling aspects. Hence, the extension of the core language to cover more meaning must be possible, but is not of paramount importance here.

Understanding a collection of UML diagrams as views onto a common underlying model enables us to unambiguously define the meaning of each diagram type in terms of the underlying core modelling language and thus automatically creates strong consistency constraints between the different diagram types. UML's current efforts to define consistency constraints using natural language and OCL must appear as sporadic ad-hoc attempts when compared to the possibilities opened up by the above described approach.

It is illusory to pretend that our core language could be equally useful for typeless languages such as SELF; too much of its expressiveness builds on the declaration of types and the associations between them. Also, its use for languages without type checking (such as SMALLTALK) will make it appear somewhat overequipped, since much of the information introduced by its models (mostly type information) will not be translated into a program. And yet, it is not a language specifically tailored for statically typed languages such as JAVA and C++; it is defined solely on formal grounds and makes no commitments to language specific features (even though we made it one of our design goals that the relationship to the pragmatics of programming is as straightforward as possible).

6 Conclusion

We have presented a replacement for UML's current core semantics that integrates static and dynamic modelling aspects into a core modelling language, built on a minimal set of concepts. We have specified the abstract syntax of the new core using a semiformal language based on set theory, and provided a set of mapping rules from UML's current concrete syntax to the abstract. We fully realize that our work can only be a first round towards a new, simplified and consistent language definition, but are optimistic that our approach will prove sustainable.

References

[1] KH Bläsius, U Hedtstück, CR Rollinger (eds) *Sorts and Types in Artificial Intelligence* Lecture Notes in Artificial Intelligence 418 (Springer 1989).

[2] OMG *Unified Modeling Language Specification* Version 1.1 (www.omg.org, September 2001).

[3] F Steimann "On the representation of roles in object-oriented and conceptual modelling" *Data & Knowledge Engineering* 35:1 (2000) 83–106.

[4] F Steimann "A radical revision of UML's role concept" in: *UML 2000* Proceedings of the 3rd International Conference (Springer 2000) 194–209.

[5] F Steimann *Formale Modellierung mit Rollen* Habilitationsschrift (Universität Hannover, 2000).

[6] F Steimann "Role = Interface: a merger of concepts" *Journal of Object-Oriented Programming* 14:4 (2001) 23–32.

[7] P Stevens "On Associations in the Unified Modelling Language" in: *UML 2001* Proceedings of the 4th International Conference (Springer LNCS 2185, 2001) 361–375.