Rigorous EBNF-based Definition for a Graphic Modeling Language

Yong Xia, Martin Glinz
Institut für Informatik der Universität Zürich
Winterthurerstr. 190, CH-8057 Zurich, Switzerland
{xia, glinz}@ifi.unizh.ch

Abstract

Today, the syntax of visual specification languages such as UML is typically defined using meta-modelling techniques. However, this kind of syntax definition has drawbacks. In particular, graphic meta-models are not powerful enough, so they must be augmented by a textual constraint language.

As an alternative, we present in this paper, a text-based technique for the syntax definition of a graphic specification language. We exploit the fact that in a graphic specification language, most syntactic features are independent of the layout of the graph. So we map the graphic elements to textual ones and define the context-free syntax of this textual language in EBNF. Using our mapping, this grammar also defines the syntax of the graphic language. Simple spatial and context-sensitive constraints are then added by attributing the context-free grammar. Finally, for handling complex structural and dynamic information in the syntax, we give a set of operational rules that work on the attributed EBNF.

We explain our syntax definition technique by applying it to the modelling language ADORA which is being developed in our research group. We also briefly discuss the application of our technique to the syntax definition of UML.

At last we mention the advantages of our method over the metamodeling techniques.

1. Introduction

Graphic modeling languages for expressing requirements and design have been used since about 25 years, progressing from SADT [15] and Structured Analysis [1] over early object modeling languages such as OMT [16] to UML [13] which is the dominating modeling language of today. In our research group in Zurich we also have developed a modeling language for requirements and architecture called ADORA [3].

A proper definition of the syntax of such languages is crucial both for their use by humans and for building appropriate tools. In contrast to textual languages, where syntax definition is well understood, the definition of the syntax for a graphic language is not an easy task. Early graphic modeling languages such as SADT or dataflow diagrams in Structured Analysis did not have any formal definition at all. The language was informally defined using natural language and tables of the graphic symbols. On the other hand, there were textual modeling languages that had a clean formal definition, for example RML [4] or ESPRESO [9] and its successor, SPADES [10].

In order to achieve a more formal definition of the syntax of graphic modeling languages, three approaches have been tried: metamodeling, graph grammars and abstract graphs. We briefly review these three approaches.

Metamodeling emerged with the design of entity-relationship-based modeling languages [5]. The principal idea is to define the syntax of an entity-relationship-based language by a model that is expressed either in the same language or in a simpler, also entity-relationship-based language. (Principally speaking, every definition of a modeling language is a metamodel, including definitions by grammars or by formal logic. However, the term 'metamodel' is typically restricted to the meaning described above.) Today, UML is the most prominent representative of this style of syntax definition, using an elaborate metamodeling technique with four layers of metamodels. The idea of metamodeling is appealing, because (i) the syntax definition diagrams are—at least in principle—easy to read for humans, and (ii) when constructing tools, the metamodel can directly be used as a database schema for the repository of the tool and/or as a common data exchange model.

However, syntax definition by metamodeling has considerable drawbacks in practice. Firstly, for big languages such as UML, the metamodels become so large that understanding and navigating through them becomes a difficult and tedious task for humans. UML makes the metamodels more compact by extensively using inheritance. However, this makes the comprehension and navigation problem even worse [6]. Secondly, graphic metamodels are not powerful enough to express even the context-free syntax of a graphic language completely. So the metamodel must be augmented by constraints that must be expressed separately in a (typically textual) constraint language.

Graph grammars have been proposed for defining the syntax of visual programming languages. The idea is to interpret graphic models as mathematical graphs and to use graph transformation rules to express the rules for constructing correct diagrams. Interpreting diagrams written in a visual language as graphs is appealing because the graphs preserve the information given by the graphic layout of the diagrams which is syntactically relevant to a large extent in visual programming languages [12]. However, in graphic languages for modeling requirements and design, layout is much less syntax-sensitive than in visual programming languages. Moreover, only a restricted class of graphs can be described by graph grammars such that a parser can be constructed directly and easily from the grammar. In Section 7, we will compare string grammars, graph grammars and our approach to syntax definition. This discussion will provide further arguments why graph grammars are not a good fit for defining the syntax of typical graphic modeling languages.

Abstract graphs [2] is an approach that first transforms concrete diagrams written in a graphic modeling language into an abstract, topology-preserving graph with typed nodes and edges. This graph is interpreted as a metamodel of the language. Syntax information that cannot be expressed in this metamodel has to be added using a logic-based constraint language. This is a quite attractive technique when the focus lies on getting a sound syntactical basis for the construction of tools. However, the syntax definition is difficult to read for humans.

In this paper, we explore an alternative path for the syntax definition of graphic modeling languages. When confronted with the problem of defining the syntax of our own modeling language ADORA [3] [8] precisely, we were both frustrated by the problems and inconveniences of the existing techniques described above and appealed by the elegance and simplicity of EBNF-(extended Backus-Naur Form)-based syntax definition for textual programming languages. Hence we decided to try whether it would be possible to define the syntax of a graphic requirements and architecture modeling language such as ADORA using an EBNF-style grammar.

We exploit the fact that most syntactic features of a graphic specification language are independent of the topology of the graph. Topology-independent elements of a graphic language can easily be mapped onto equivalent elements of a textual language. So the essence of our approach is to define this mapping and to define the resulting textual language rigorously with an EBNF grammar. We found that this works quite well for the context-free syntax of the language, because the topology-dependent features of a graphic modeling language are typically context-sensitive (nesting of model elements, for example). We define the context-sensitive syntax of the language in two steps. Static context-sensitive constraints are expressed by attributing the EBNF grammar. For the remaining complex and dynamic context-sensitive constraints, we define a set of operational rules that work on the attributed EBNF.

Although our syntax definition technique was developed for the definition of the ADORA language, it applies to other graphic modeling languages as well. At the end of this paper, we will briefly discuss the applicability of our approach to UML.

Using a BNF-style grammar for defining the syntax of modeling languages is not new. It has been successfully applied before to textual modeling languages [9], including textual languages that also have a graphic representation of models or parts thereof (primarily for browsing purposes) [10] [7]. There are also tools that represent graphic metamodels internally with a textual language and define the context-free syntax of this meta language with a BNF grammar.

The contribution of our approach is threefold:

- We demonstrate that an EBNF-based syntax definition of a graphic modeling language is feasible for modeling languages that have been designed as graphic languages from the beginning (in contrast to textual languages having an additional graphic representation).
- We achieve an elegant and concise syntax definition with a clear separation between context-free syntax, context-sensitive syntax and semantics. Moreover, we introduce a new way of defining context-sensitive syntax with operational rules.
- We express the syntax definition in a unified framework whereas the currently dominating metamodeling approaches require a combination of a (typically graphic) syntax definition language and a (typically textual) constraint language.

The rest of the paper is organised as follows. In section 2 we present the basic concepts of our approach. Section 3 gives a short introduction to ADORA , because this language will be used as a sample graphic modeling language in this paper. In section 4 we demonstrate the context-free syntax definition in EBNF, using the core of ADORA as an example. Section 5 discusses the definition of context-sensitive syntax. Firstly, we demonstrate the definition of simple static constraints by attributing the context-free grammar which was introduced in the previous section. Secondly, for expressing dynamic constraints we construct a set of operational rules that work on the attributed EBNF. Section 6 briefly investigates the applicability of our approach to the syntax definition of UML. We conclude with a discussion of achievements, limitations and plans for future research.

2. Basic concepts

In this section, we introduce the four basic concepts of our syntax definition method.

Graphics to text mapping. Let GML be a given graphic modeling language the syntax of which we want to define. We first define a textual modeling language TML such that there exists an isomorphic (i.e. one-to-one) mapping from every language element of GML to a corresponding language element of TML. The mapping is defined in tabular form (cf. Figure 3). As graphic modeling languages for requirements and architecture contain quite few elements where the graphic layout is syntactically relevant, we can successfully construct such a textual language and mapping. Having defined TML and an isomorphic GML-TML mapping, every formal definition of the syntax for TML also defines the syntax of our original language GML.

Context-free syntax. We use EBNF (extended Backus-Naur form) for the definition of the context-free syntax of the textual language TML. (E)BNF is a proven syntax definition technique which is easy to understand for humans and where we have mature and efficient techniques for constructing parsers.

Static context-sensitive syntax. Graphic modeling languages typically have a syntax that is partially context-sensitive. For example, the rule that an aggregation hierarchy must be acyclic is context-sensitive. As long as these rules pertain to static syntax, (i.e. they constrain the static structure of models), they can easily be defined by attributing the EBNF grammar. The attributes of an EBNF production rule are logic formulae that must evaluate to true when the rule is evaluated. Attribute parameters may be passed from rules to subrules and vice-versa.

Dynamic context-sensitive syntax. Finally, the syntax of a graphic modeling language may also contain dynamic rules. In ADORA for example, a diagram may be modified by adding more detail to the representation of an object. In this case, the graphic representation of the relationships that this object has with other objects also must be modified. Formally describing such consistency rules is rather difficult, because they cannot be modeled with attributed grammars. We use operational rules similar to those used for operational definition of language semantics [17] to express such dynamic syntax constraints.

3. A short introduction to ADORA

In this section, we give a brief introduction to ADORA, as we will use the ADORA language as a sample graphic modeling language in the rest of this paper. ADORA is a modeling technique for requirements and software architecture that is being developed in our research group [3] [8]. The acronym stands for Analysis and Description of Requirements and Architecture. Figures 1 and 2 (taken from the specification of a distributed heating control system) give an impression how ADORA models look like. At the first glance, ADORA diagrams look similar to UML diagrams. However, there are fundamental differences between ADORA and UML [3]. Below, we summarize the distinguishing features of ADORA.

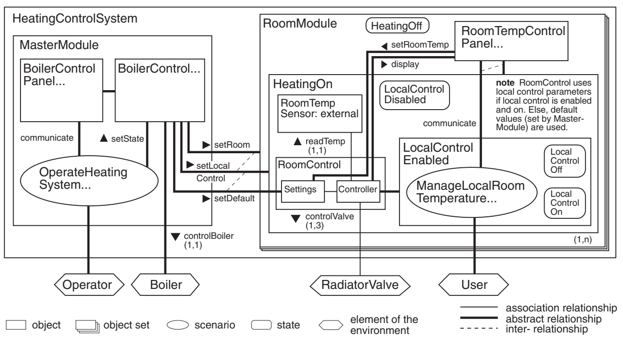
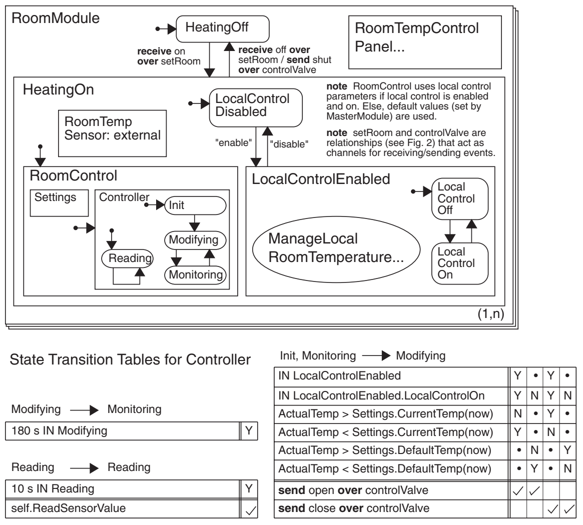


Figure 1. An ADORA view of the heating system: base view + structural view + context view

Using abstract objects (instead of classes) as the basis of the model. Class models are inappropriate when more than one object of a class and/or structural nesting of objects has to be modeled. Therefore, ADORA uses *abstract, prototypical objects* instead of classes as the conceptual core of the language. For example, in sample heating control system (see Figure 1), there is a single Master Module, but multiple room modules. In ADORA, we model these entities as abstract objects and thus can make these cardinalities immediately visible. Moreover, the Boiler Control Panel in Master Module and the Room Control Panel in the Room Module may have the same type. Hence, they would not be distinguishable in a class model, while with abstract objects, we can model them separately and place them where they belong.

Hierarchical decomposition. ADORA systematically uses hierarchical decomposition for structuring models. With the use of abstract objects, abstraction and decomposition mechanisms can easily be introduced into the language. We recursively decompose objects into objects (or other elements, like states). So we have the full power of object modeling on all levels of the hierarchy and only vary the degree of abstractness: objects on lower levels of the decomposition model small parts of a system in detail, whereas objects on higher levels model large parts or the whole system on an abstract level.

Integration of all aspects of the system in one coherent model. An ADORA model integrates all modeling aspects (structure, data, behavior, user interaction . . .) in one coherent model. This allows us to introduce strong rules for consistency and completeness of models, reduces redundancy, and makes the model construction more systematic.
Using an integrated model does of course not mean that everything is drawn in one single diagram. From the integrated model, we can generate *views* pertaining to a given *aspect*. The so-called *base view* of an ADORA model consists of the hierarchical structure of objects only. Aspect views are generated by combining the base view with all information that is relevant for the selected aspect. For example, Figure 1 shows the structure view (which shows the static structure of the system by combining the base view with directed relationships between objects/object sets) and the context view (which shows all actors and objects in the environment of the modeled system and their relationship with the system) of the whole heating control system. Figure 2[1] shows the behavior view (which shows the dynamic behaviors of the system by combining base view with a statechart-based state machine hierarchy) of the Room Module in our heating control system.


RoomModule
HeatingOff
receive on over setRoom
receive off over setRoom / send shut over controlValve
RoomTempControl Panel...
HeatingOn
LocalControl Disabled
note RoomControl uses local control parameters if local control is enabled and on. Else, default values (set by MasterModule) are used.
note setRoom and controlValve are relationships (see Fig. 2) that act as channels for receiving/sending events.
RoomTemp Sensor: external
"enable"
"disable"
RoomControl
Settings
Controller
Init
Modifying
Reading
Monitoring
LocalControlEnabled
Local Control Off
Local Control On
ManageLocal RoomTemperature...
(1,n)

State Transition Tables for Controller

Modifying $\longrightarrow$ Monitoring

180 s IN Modifying	Y

Reading $\longrightarrow$ Reading

10 s IN Reading	Y
self.ReadSensorValue	✓

Init, Monitoring $\longrightarrow$ Modifying

IN LocalControlEnabled	Y	•	Y	•
IN LocalControlEnabled.LocalControlOn	Y	N	Y	N
ActualTemp > Settings.CurrentTemp(now)	N	•	Y	•
ActualTemp < Settings.CurrentTemp(now)	Y	•	N	•
ActualTemp > Settings.DefaultTemp(now)	•	N	•	Y
ActualTemp < Settings.DefaultTemp(now)	•	Y	•	N
send open **over** controlValve	✓	✓		
send close **over** controlValve			✓	✓

Figure 2. A partial ADORA model of the heating system: base view + behavior view

1 In this figure, the triggering events and triggered actions/events are expressed in tabular form to make the model more readable. They can be translated into the equivalent traditional form as an adornment of the state transition arrows in the diagrams.

4. Definition of context-free syntax

Using the ADORA language as an example, we demonstrate in this section how we define the context-free syntax of a graphic modeling language in our approach. We start with some notational definitions. Then we define the graphics-to-text mapping and present the syntax definition of the resulting textual language in EBNF.

4.1. Meta symbols and notations

First we introduce some notations and meta symbols which will be used in our EBNF syntax. Let x, y, z be nonterminal symbols.

- *[x]* denotes zero or one occurrence of x
- $\{x\}$ denotes zero or many occurrence of x
- $\{x\}_1$ denotes one to many occurrence of x
- $x \mid y$ means either x or y
- $z ::= \ldots x : y \ldots$ is a shorthand notation for the following two rules: $z ::= \ldots x \ldots$ and $x ::= y$.

4.2. The mapping

The mapping from the given graphic modeling language (the ADORA language in our case) to an equivalent textual language is defined in tabular form (Figure 3). Every graphic model element is uniquely mapped to a corresponding textual phrase.

For example, a rectangle with a string in the upper left edge in ADORA represents a single abstract object with its name given by the string. This element is mapped to the textual phrase **object** name **end**. The graphic element in the third row of the mapping table contains EBNF rules, for example $\{system\_object\}$. Such a rule means that at this place any graphic elements are allowed that translate to textual phrases satisfying the rule.

A naive mapping of the language elements may result a table with infinite lines. By introducing some sophisticated mapping techniques (e.g. using the defined EBNF grammar categories in the graphical notation to summarize the combinations of basic graphic elements), we can map the whole language in a table with some dozens of lines. Please refer to [18] for the details.

Having done this mapping, we have a textual language that is equivalent to the original graphic language.

4.3. EBNF syntax

Due to limited space, only the core part of the ADORA EBNF syntax will be listed in this section. The complete syntax definition can be found in [18].

Specification
specification ::= { specification_fragment }
specification_fragment ::= { model_element }

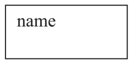
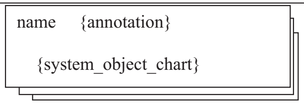
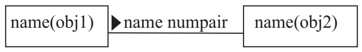
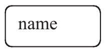
Graphic element	Equivalent textual element
	**object** name **end**
...	...
	**object_set** name {annotation} [**contains** {system_object_chart}$_1$] **end**
...	...
	**association** name numpair from obj1:object **to obj2:object** end
	**state** name
	start_state
...	...

Figure 3. An Excerpt of the ADORA Mapping Table

model_element ::=
 object | state | scenario | relationship | annotation
Types
Literal ::=
 FloatingPointLiteral | IntegerLiteral | CharacterLiteral |
 StringLiteral | BooleanLiteral | NullLiteral
IntegerLiteral ::=
 NonNegativeIntegerLiteral | NegativeIntegerLiteral
numpair ::=
 NonNegativeIntegerLiteral "," NonNegativeIntegerLiteral
name ::= identifier
...
Expression
...
Operator
...
Annotation
annotation ::= ...
Objects/States/Scenario declarations
object ::= object_i | singleton_object_e | object_set_e

system_object ::= object_i | state | scenario |
  singleton_object_e | object_set_e

object_i ::= singleton_object | object_set
singleton_object ::= **object** name [obj_body] **end**
singleton_object_e ::= **object** name ": external" **end**
object_set ::= **object_set** name [obj_body] **end**
object_set_e ::= **object_set** name ": external" **end**
env_object ::= **element_of_the_environment** name
obj_body ::=
 {annotation} [**attributes** {attribute}$_1$]
 [**operations** {operation}$_1$] [**contains** {system_object}$_1$] attribute ::= ...
operation ::= ...
state ::= pure_state | start_state
pure_state ::= **state** name [**contains** {state}$_1$] **end**
start_state ::= **start_state end**
scenario ::= **scenario** name [scenario_body] **end**
scenario_body ::= ...
Relationship/Transition declarations
relationship ::=
 association | abstract_relationship | interrelationship
association ::= **association** name numpair
  from obj1:object **to** obj2:object **end**[2]

abstract_relationship ::=
 abstract_relationship {name}
 connecting obj1:object **and** obj2:object **end** |
 abstract_relationship {name}
 connecting obj:object **and** sc:scenario **end** |
 abstract_relationship {name}
 connecting obj:object **and** e:env_object **end** |
 abstract_relationship {name}
 connecting sc:scenario **and** e:env_object **end**

interrelationship ::= **interrelationship**
  sub a:association
  super ar:abstract_relationship
  end |
  interrelationship
  sub ar1 : abstract_relationship
  super ar2 : abstract_relationship
  end

transition ::= **transition** t_condition/t_action
  from obj1:object_i **to** obj2:object_i **end** |
  transition t_condition/t_action
  from obj_f:object_i **to** s_t:pure_state **end** |
  transition t_condition/t_action
  from s_f:pure_state **to** obj_t:object_i **end** |
  transition t_condition/t_action
  from ss:start_state **to** obj:object_i **end** |
  transition t_condition/t_action
  from s1:pure_state **to** s2:pure_state **end** |
  transition t_condition/t_action
  from ss:start_state **to** s:pure_state **end**

t_condition ::= ...

t_action ::= ...

...

4.4. Remarks

Below, we make some remarks about the EBNF syntax definition in the previous subsection.

- The non-terminal symbol names in our EBNF definition do not not always have the same meaning as these names have in the ADORA language. For example, the name "object" in the EBNF grammar includes the types *object, object set, external object, external object set* in the ADORA language. To avoid confusion, all references to symbol names in the EBNF grammar are underlined in the following text (e.g. system_object).
- Due to space limitations, the detailed definition of expression, operator, annotation, attribute, operation, t_condition, t_action, etc. are not presented here, as most of them are defined in the same way as in other textual specification/programming languages, such as Z, Java, etc.
- The meaning of abstract relationships and interrelationships are explained in section 5.3.
- In a system specification, the names of objects and object sets are unique. However, the names of relationships, states, etc., are not unique. We need to define extra attributes, such as "associationID", to identify those elements. For simplicity, we assume in this paper that these elements are also identified by their name.

2 In a pure context-free grammar, it would obviously be simpler to state this rule just as: association ::= **association** name numpair **from** object **to** object **end**. However, when we later attribute this rule for expressing the context-sensitive syntax, we need to denote the identifiers obj1 and obj2. The same is true for other rules, for example those defining transitions.

- *Abstract syntax vs. concrete syntax.* The presented EBNF grammar is an abstract syntax, which is used for human beings to master the language. It needs to be further optimized and concretized for the purpose of construction of a parser/compiler (e.g. adding some extra reference elements in the grammar to enhance the parsing efficiency). Nevertheless, unlike the metamodeling technique, in which a concrete syntax for the compiler design can only be derived from the semantic interpretation of the metamodel (an abstract syntax), the derivation from the abstract syntax to the concrete syntax in the EBNF-based approach is only a process of optimization, and hence much easier and more straightforward.

5. Definition of context-sensitive syntax

Even for textual programming languages, a context-free grammar is in most cases not powerful enough for expressing the syntax of the language completely. In a graphic modeling language, we have additional spatial layout constraints that are also context-sensitive.

We define the context-sensitive syntax in a way that is well-proven for EBNF grammars: we attach constraint attributes to the EBNF grammar rules. Every attribute is a logic formula. Whenever an EBNF rule is evaluated, all attributes of this rule must be true.

Attributes are well suited for the definition of *static* context-sensitive syntax constraints. However, in particular in the ADORA language, we also have *dynamic* context-sensitive syntax constraints that pertain to transformations of an ADORA model into another representation such that syntactic correctness is preserved. Describing such constraints with an attributed grammar is hard or even impossible. Therefore, dynamic constraints will be defined with operational rules (see Section 5.3).

In the ADORA language, the hierarchical structure of objects is the most important source of static context-sensitive syntax constraints. In order to express these constraints, we introduce the concept of a *pretrace* in the following subsection. Using this concept, we will then present selected attributed EBNF grammar rules.

5.1. Capturing hierarchical structure

Every object that is part of a hierarchical structure is hierarchically embedded in one or more other objects. For example, in Figure 4, A is embedded in A', A'' and in A''', A' is embedded in A'' and in A''', etc. Only the outmost object A''' is not embedded. In order to get rid of this exception, we assume that every object is always embedded in itself.

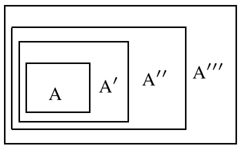


Figure 4. A sample object hierarchy

Let $A \Rightarrow B$ denote the fact that A is *directly embedded* in B; that means there exists no X such that $A \neq X \neq B$ and $A \Rightarrow X \Rightarrow B$.

Now we define the *pretrace* of a $\underline{\text{system\_object}}$ X to be the ordered set [3] of model elements that X is embedded in. Formally, we use a simple recursive definition:
if exists a $\underline{\text{system\_object}}$ B such that $X \Rightarrow B$ then $X.pre = \{X, B.pre\}_{ordered}$, otherwise $X.pre = \{X\}$.

For example, for the objects of Figure 4 we have: $A.pre = \{A, A', A'', A'''\}_{ordered}$, $A''.pre = \{A'', A'''\}_{ordered}$ and $A''' = \{A'''\}$.

$\supset$ and $\supseteq$ are the usual mathematical subset relations. Due to the definition of pretraces, $A.pre \supset B.pre$ means that A is embedded in B. Additionally, we define a special subset relation $\sqsupset$ for pretraces: $A.pre \sqsupset B.pre$ if and only if $A \Rightarrow B$.

In the example of Figure 4, we have
$A.pre \sqsupset A'.pre, A'.pre \sqsupset A''.pre, A''.pre \sqsupset A'''.pre.$

We call the pretraces of two $\underline{\text{system\_objects}}$ A and B to be *hierarchically equivalent* if and only if there exists another $\underline{\text{system\_object}}$ X where both A and B are directly embedded in, or both A and B are outmost objects that are not embedded at all. In formal terms, we write:
Hierarchical equivalence. $A.pre \simeq B.pre$ iff
$\exists X : \underline{system\_object} \bullet (A.pre \sqsupset X.pre \wedge B.pre \sqsupset X.pre)$
$\vee (A.pre = \{A\} \wedge B.pre = \{B\})$

The intuitive meaning of $A.pre \simeq B.pre$ is that either are A and B on the same hierarchical level within a common $\underline{\text{system\_object}}$ or they are both not embedded (except in themselves). The *pretrace* of a $\underline{\text{system\_object}}$ includes this object itself. Although this makes the definition of *hierarchical equivalence* more complex, it greatly simplifies the definition of the context-sensitive syntax of ADORA in later sections.

5.2. Attributing the EBNF grammar

The static context-sensitive syntax can now be defined by attributing the EBNF grammar (written with ***Boldface text in Italic shape***). The attributes of an EBNF grammar rule must be true whenever this rule is evaluated. In this paper, we present two typical examples only. The first example describes two context-sensitive constraints for associa-

3 *Ordered sets* are also called *lists* in the literature on formal methods.

tions, the second one a constraint for state transitions. In both cases the constraints are needed to ensure the hierarchical (de-)composability of ADORA models. The rest constraints can be found in [18].

In the declaration of an *association*, the constraint is:

```
association ::= association name numpair
                from obj1:object to obj2:object
                Constraints:
                   $obj1.pre \not\supset obj2.pre$
                   $\wedge$
                   $obj2.pre \not\supset obj1.pre$
                end
```

This means that if an object C is embedded in another object A (it can be one level embedded or several levels embedded), an association connecting A and C is not allowed. In Figure 5, $r1$ is syntactically wrong and $r2$ is syntactically correct. The EBNF rule for *transitions* in the syntax of the ADORA language needs six attributes to specify all the constraints. Here we present one case of this rule where a transition leads from a pure state to an object (which has an internal state). Intuitively, the constraint is that a state transition may cross state boundaries (as it is usual in statecharts), but must not cross object boundaries except the boundary of the destination object.

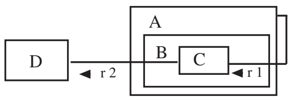


Figure 5. Context-sensitive syntax of associations: r1 is wrong, r2 is correct

```
transition::= ...
     ...|
          transition t_condition/t_action
          from s_f:pure_state to obj_t:object_i
          Constraints:
          $\exists$ S:state $\bullet$ $S \in s\_f.pre \wedge$
          $S.pre \simeq obj\_t.pre \vee S.pre \sqsupset obj\_t.pre$
          ...
```

Stated in natural language, the constraint says that there must exist a state S in the pretrace of the source state (with S being identical to the source state as a special case) such that S is either directly embedded in the destination object or S is on the same hierarchical level as the destination object (i.e. their pretraces are hierarchically equivalent). This constraint is equivalent to the intuitive rule stated above. In the example of Figure 6, transitions $t1$, $t2$ and $t3$ are syntactically correct, while transition $t4$ is syntactically wrong.

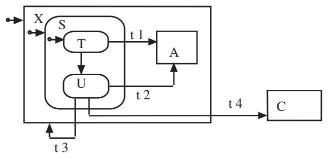


Figure 6. Context-sensitive syntax of state transitions. Transitions t1, t2, t3 are legal, t4 is illegal.

5.3. Operational rules

As we already mentioned at the beginning of section 5, there are complex context-sensitive constraints that are hard or even impossible to express with constraint attributes. In ADORA, a particular problem of this kind is the syntactic correctness of diagrams that show a (partial) view of a larger underlying model. In contrast to languages such as UML, the syntactic correctness of an ADORA diagram not only depends on the information given in the diagram, but also on information which is present in the model, but not visible in the particular diagram. At a first glance, this may look strange. However, it is a quite important feature if we want to guarantee consistency between diagrams as well as between the overall model and a diagram that shows a particular view of the model.

Example. Suppose we have a quite simple model of a system S consisting of objects P, Q, X, Y and an association r from P to Q (c.f. Figure 7c). Furthermore, let P be embedded in X and Q in Y. Now assume that we want to draw an overview diagram which shows only the overall system and its two main components, X and Y. If we would draw this diagram in the way shown in Figure 7a, we would suppress two important facts:

- X and Y have inner components
- X and Y may exchange information (due to the relationship between P and Q).

In this situation, ADORA requires us to model hints that point at suppressed information (Figure 7b):

- Trailing dots are added to the names of X and Y to indicate that these objects contain other objects that are not shown on the diagram. We also call these trailing dots an *is-partial indicator*.
- A so-called abstract relationship (thick line) is drawn between objects X and Y to indicate that at least one relationship from an inner element of X to an inner element of Y is suppressed.

When we expand this diagram to display the complete model (Figure 7c), the syntax must be changed: the dots and the abstract relationship disappear.

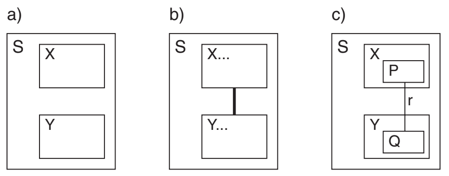


Figure 7. Syntax of ADORA diagrams that show partial models only. Diagrams b) and c) are syntactically correct representations of the example model given above in the text; diagram a) is syntactically wrong.

In order to define such dynamic syntax constraints, we introduce *operational rules*. The logical structure of these rules is similar to the rules being used in the definition of operational semantics for programming languages. However, we use a different notation in this paper in order to make the rules easier to read for humans. Due to space limitations, we present only a few rules here. The complete definition can be found in [18]. Before we can state these rules, we must define some syntactic functions.

5.3.1. Syntactic functions. We need the following syntactic functions for determining the visibility of model elements in a given partial ADORA model (typically represented in a diagram).

Let M be an ADORA model with X, Y, Z : $\underline{model\_element}$, A, B, C, D : $\underline{object}$, r : $\underline{association}$, u : $\underline{abstract\_relationship}$ and let Γ be a partial view of M.

- $visible(X, \Gamma)$ is a boolean function that is true if and only if X is visible in Γ.
  For a set of model_elements, $visible(\{X, Y, Z\}, \Gamma)$ means $visible(X, \Gamma) \wedge visible(Y, \Gamma) \wedge visible(Z, \Gamma)$.
- $hidden(X, \Gamma)$ is a boolean function that is true if and only if X is not visible in Γ. This function is defined for convenience only. By definition holds $hidden(X, \Gamma) = \neg visible(X, \Gamma)$ for all X and Γ.
- $partial(A, \Gamma)$ is a boolean function that is true if and only if the name of A in Γ is followed by three dots.
- The meaning of the $\subset$ and $\sqsubset$ operators is extended to relationships as follows.
  - $r(A, B) \subset u(C, D)$ if and only if $A.pre \supset C.pre$ and $B.pre \supset D.pre$.
  - $r \sqsubset u$, if and only if $r \subset u$ and there is no u_1 in Γ, such that $r \subset u_1 \subset u$.

5.3.2. General format of rules. Every operational rule has the following format.

Rule *rule-name* $(parameters)$

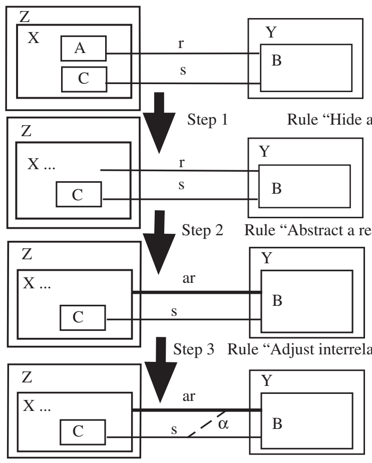


Figure 8. The process of hiding an object from an ADORA diagram. The original and the final diagram are syntactically correct, the intermediate ones are not.

Given: $M; \Gamma; model\ elements$
Condition: $predicate\_pre$
Assertion: $predicate\_post$
Next Rule: *rule-name* $(parameters)$

Rule names need not be unique; we may have more than one rule with the same name but with different conditions. A rule is interpreted as follows: for any ADORA model M that contains the given $model\ elements$ and has a partial model Γ such that $predicate\_pre$ is true, the application of the rule modifies Γ so that $predicate\_post$ becomes true. The application of the rule does not modify anything that is not specified in $predicate\_post$. If the **Next Rule** field contains a name, the rule(s) matching this name must be applied next in order to transform a syntactically correct view Γ eventually into a new view Γ_1 which is also syntactically correct. Rule execution stops when the **Next Rule** field is empty or when the conditions ($predicate\_pre$) of all matching rules are false. Parameters may be used to transfer information from a rule to the next one.

5.3.3. Rules for hiding an object. In this subsection, we present the operational rules that describe the process of making an ADORA diagram more abstract by hiding one of its objects. The process is illustrated in Figure 8. In this example, three rules must be applied consecutively in order to transform an syntactically correct diagram into a more abstract one that is again syntactically correct.

The dashed line in the bottom diagram of Figure 8 connecting the association s and the abstract_relationship ar is a so-called *interrelationship*. It denotes the fact that ar is not only an abstraction of the hidden association r, but also of s.

Hide an object

Rule	Hide an object (A)
Given:	$M; \Gamma;\ X : \underline{object\_i}; A : \underline{object}$
Condition:	$(A.pre \sqsupset X.pre) \wedge visible(\{A, X\}, \Gamma) \wedge$
	$(\neg\exists Y : \underline{object} \bullet Y.pre \supset A.pre)$
Assertion:	$hidden(A, \Gamma) \wedge partial(X, \Gamma)$
Next Rule:	Abstract a relationship (A)

... ...

Abstract a relationship

Rule	Abstract a relationship (A)
Given:	$M; \Gamma;\ X : \underline{object\_i}; A, B : \underline{object};$
	$r(A, B) : \underline{association}$
Condition:	$(A.pre \sqsupset X.pre) \wedge$
	$hidden(A, \Gamma) \wedge visible(\{B, X, r(A, B)\}, \Gamma)$
Assertion:	$hidden(r(A, B)) \wedge \exists ar : \underline{abstract\_relationship} \bullet$
	$(ar(X, B) \in \Gamma) \wedge\ visible(ar, \Gamma) \wedge$
	$\forall z : \underline{abstract\_relationship}$
	$\neg\exists\alpha : \underline{interrelationship} \bullet\ \alpha(r, z) \in \Gamma$
Next Rule:	Adjust interrelationships (ar)

... ...

Adjust interrelationships

Rule	Adjust interrelationships (ar)
Given:	$M; \Gamma;\ s : \underline{association}; ar : \underline{abstract\_relationship}$
Condition:	$(s \sqsupset ar) \wedge visible(s, \Gamma) \wedge$
	$\neg\exists\alpha : \underline{interrelationship} \bullet \alpha(s, ar) \in \Gamma$
Assertion:	$\exists\alpha : \underline{interrelationship} \bullet \alpha(s, ar) \in \Gamma$
Next Rule:	Adjust interrelationships (ar)
Rule	Adjust interrelationships (ar)
Given:	$M; \Gamma;\ ar, z : \underline{abstract\_relationship}$
Condition:	$(ar \sqsupset z) \wedge$
	$\neg\exists\alpha : \underline{interrelationship} \bullet \alpha(ar, z) \in \Gamma$
Assertion:	$\exists\alpha : \underline{interrelationship} \bullet \alpha(ar, z) \in \Gamma$
Next Rule:	Adjust interrelationships (ar)
Rule	Adjust interrelationships (ar)
Given:	$M; \Gamma;\ s : \underline{association}; ar, z : \underline{abstract\_relationship}$
Condition:	$visible(s, \Gamma) \wedge (s \sqsupset ar \sqsupset z) \wedge$
	$\exists\alpha : \underline{interrelationship} \bullet \alpha(s, z) \in \Gamma$
Assertion:	$\neg\exists\alpha : \underline{interrelationship} \bullet \alpha(s, z) \in \Gamma$
Next Rule:	Adjust interrelationships (ar)

... ...

In the situation illustrated in Figure 8, the transformation starts with the "Hide an object" rule. According to its **Next Rule** clause, "Abstract a relationship" rules have to be executed next, with the hidden object A as a parameter. There exist several rules with this name. However, in this paper we show the sole rule which applies in our example case (i.e. where the condition is true). In the next step, "Adjust interrelationships" rules have to be executed. Three of these rules are listed in the paper. In our example, only the condition of the first of them is true. So this rule is executed. According to the **Next Rule** clause,"Adjust interrelationship" rules have to be executed again. However, in the new situation the conditions of all these rules are false. Hence, rule execution terminates and the transformation is complete.

For readers who do not want to dig into the formalisms, we give the meaning of the first rule (Hide an object) in words. The rule applies to any model M and view Γ where we have objects A and X such that (i) A is directly embedded in X, (ii) there are no objects embedded in A, and (iii) both A and X are visible in the view Γ. After application of the rule, A is hidden from the view, and the name of object X has three dots appended.

5.3.4. Remarks. The **Given** field and the parameters of the rule define the model elements being used as variables. Note that there are no free variables in the **Condition** predicate. Parameter variables are externally bound; the other variables are bound to the existential quantifier.

Here only the rules relating to a simple linear sequence of view transitions are given. Actually the real situation is much more complex: when an object is hidden (or becomes visible), depending on the original states of the surrounding objects and relationships, the resulting view may look considerably different from the originating one. The complete definition, which covers all the possible situations, has more than 30 operational rules. The strict logic definition of the rules and the execution sequence are given in [18].

5.3.5. Syntax vs. Semantics. The context-sensitive syntax and the operational rules could also be considered to be *semantics* instead of *syntax*. For example, in the UML specification documents from OMG [13], the constext-sensitive syntax mentioned above is called *static semantics*, because it specifies how an instance of a construct to be *meaningfully* connected to other instances. In some literature, operational rules are even classified as *dynamic semantics*, as they imply the meanings of some constructs (e.g. is-partial indicator and abstract relationship) and provide the theoretical foundation for specification verification and refinement. On the other hand, [11][14] think that the syntax encompasses all the compile-time aspects, while (dynamic) semantics encompasses the run-time aspects.

Both classifications are reasonable in some sense. However, only one should be used in a paper to avoid ambiguity and misunderstanding. In our approach we adopt the second classification from Frank Pagan[14].

6. Syntax definition of UML

In this paper, we selected the ADORA language as a vehicle for demonstrating our new method of syntax definition for graphic modeling languages. We did so for two reasons. Firstly, the hierarchical decomposition mechanism and the elaborate model integrity concept in ADORA are challenging problems when defining more than just the context-free syntax formally. So if our approach works for ADORA, it should also work for other, less demanding graphic languages. Secondly, we needed a precise syntax definition of ADORA as a part of our effort to develop this language.

For the definition of the syntax of UML, our method can be applied in a quite similar way as we did it for ADORA: (1) setting up a mapping table which defines an equivalent textual expression for every graphical element and constructor in UML; (2) defining the context-free syntax of UML with EBNF on the translated textual form; (3) adding context-sensitive syntax rules by attributing the context-free grammar to describe the spatial information in the UML. As

UML lacks a sound and elaborate concept for inter-diagram integrity checking, an attributed grammar should be expressive enough to describe the whole UML syntax. Defining operational rules for parts of the syntax will most probably not be necessary.

We are currently defining a part of the UML 1.4 core using our syntax definition method. As a preliminary result, we can say that our technique is simpler and more systematic than the metamodeling technique used in the OMG definition of UML 1.4 [13]. For example, when looking for a precise definition of "class" in the OMG metamodel (which implies inspecting also the inherited and associated properties of "class"), the reader has to inspect 5 metamodel diagrams and about 3 pages of OCL constraints – and the metamodel does not provide the reader any roadmap for systematically navigating through this material in order to find the relevant information. In our attributed EBNF grammar we need fewer rules and we provide a simple and straightforward navigation strategy: rule inspection starts with the rule where "class" appears on the left side. Then the reader has to inspect recursively all rules that appear on the right side of an inspected rule.

In essence, our EBNF grammar (*in a textual form*) and OMG's metamodel (*in a graphical form*) describe the structure of the language (the language constructs and the relationships among those constructs). The constraints on this language structure are described in *textual* forms both in our approach and in the OMG-UML technique: we use an attribution of the EBNF grammar and operational rules, OMG uses OCL. As we define the language structure and its constraints both in a textual form, our language definition is relatively easier to be organized in a more systematic way. For users, it is also easier to read it in a systematic way. Moreover, as our constraints are embedded in the EBNF grammar, they are more concise (no elements in metamodel diagrams need to be extra referred to) and straightforward.

7. Conclusion

Achievements. A good syntax definition for a modeling language should serve two purposes. On the one hand, it should help users to understand and master the language. On the other hand, it should help the tool developers to construct tools that support the language. As we discussed in the introduction, the existing syntax definition techniques for graphic languages, in particular the metamodeling technique, do not achieve these goals satisfactorily.

We think that our approach performs better with respect to both goals. Our EBNF-based syntax definition is a compact reference that helps system developers to understand the language and to create syntactically correct models. For the tool designer, it gives a formal and unambiguous specification of the language. Every graphic modeling language contains textual elements for modeling detail (for example, definitions of attributes and operations). While we can easily integrate the syntax definition for these parts of the language, metamodeling-based or graph-based techniques can't.

Moreover, the defined syntax of a graphical modeling language using our EBNF-based method is very similar to that of a textual specification language or programming language. Therefore, the existing *semantics* definition techniques which are used in defining semantics of textual specification or programming languages, can also be applied to a graphical modelling language with a few changes only.

We do not dismiss the other syntax definition techniques for modeling languages (in particular, metamodeling) as being bad or obsolete. Our approach is meant to be an alternative that avoids problems of the existing techniques. Neither do we claim that constructing a textual modeling language and defining its syntax with a BNF grammar are new ideas. Our contribution is to demonstrate that this technique is feasible for a complex graphic modeling language and that it can be used not just as an internal representation in a tool, but as a reference both for the human users of the language and for the tool builders.

Graph grammars extend string grammars [12] (e.g. EBNF) in the following points: (i) terminals are interpreted as two-dimensional objects; (ii) attributes are used to specify the spatial information; (iii) the one-dimensional concatenation in the production rules is extended to multi-dimensional concatenation; and (iv) the functionalities of the elements in the production rules are augmented. In a sense, our EBNF based definition method can also be somehow thought as an extended string grammar: the terminals in our grammar are actually also the two-dimensional object, after reverse mapping from the textual modeling language to the graphical modeling language. The difference is: in most graph grammars, those two-dimensional objects are basic graphical elements, such as straight line, arc, etc. In our method, those "two-dimensional objects" are basic model elements, such as object, association, etc. That is to say, the abstract levels of the terminals in the two approaches are different. As our grammar is not intended to be used in fields such as image processing, our interpretation is proper for our purposes. Furthermore, as our interpretation makes a graphic modeling language nearly one dimensional, attributes in our approach are mainly used to specify context sensitive syntax (static semantics), instead of spatial information. The third and fourth extensions in graph grammars are not necessary in our approach at all. In a word, our approach extends the string grammar in a suitable way to be expressive enough to handle the typical graphic modeling languages. At the same time, it is far from the complexity of graph grammars, which are certainly more expressive (the extra expressiveness is not useful for our purposes) but much more difficult to use and understand.

Another important contribution of this paper is our innovative way of defining dynamic context-sensitive constraints with operational rules. We demonstrate that these operational rules are a clean and elegant way of dealing with hierarchy and with partial models and that they nicely integrate with our grammar-based definition of context-free syntax. The advantage of operational rules over a declarative set of constraints is that quite complex syntactic rules can be expressed as a series of sequential steps, each of which being easily readable and implementable.

Last but not least, a formal and complete syntax definition for wide spectrum graphic modeling languages (e.g. ADORA and UML), which are a set of tight and loose coupled simple graphic modeling languages (e.g. statechart, MSC, Petri Net), is a significant work worthy of much more attention than what the existing researches are devoted to. After a *careful* survey, we *do not* find *any works* similar to our approach. Metamodeling is a semi-formal approach even for the context-free syntax definition. In other literature, the works on syntax definition are only for simple graphic modeling languages, and most of them are just by-products of semantics definition. This is not acceptable for a wide spectrum graphic modeling language with a rich syntax, where integration conflicts of different sub-languages must be eliminated and constraints on the numerous constructs have to be precisely specified.

Limitations. When used by humans, our syntax definition technique requires mental effort for translating graphic concepts into textual ones and vice-versa, using the mapping table. Therefore, our approach is limited to languages where this mapping is easy and straightforward. As we discussed in the paper, there is always an easy mapping when the graphic layout of the diagrams has little or even no syntactic relevance. With few exceptions, this is the case for requirements and architecture modeling languages. In this family of languages, only sequence diagrams have a strongly layout-dependent syntax. Moreover, other techniques such as metamodeling also require substantial mental effort for combining a graphic language (the metamodels) with a textual language (the constraints).

Future work. With the development in the software engineering field, the ADORA language will also evolve. Existing constructs will be modified and new concepts will be supported. Correspondingly, the language definition will not be frozen, but evolve too. We also plan to develop a parser for the ADORA language based on our syntax definition and to integrate it into our prototype ADORA tool. What is more, based on the current language definition (especially on the operational rules, which handle hierarchical structure of the system), a refinement calculus is proposed. A set of verification rules can also be built on our formally defined syntax. These works make software development using ADORA with different degrees of formality (from semi-formal to totally formal) possible.

References

[1] T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

[2] J. Ebert, R. Süttenbach, *An OMT Metamodel*, Technical Report 13, University of Koblenz, 1997.

[3] M. Glinz, S. Berner, and S. Joos, "Object-oriented modeling with ADORA". *Information Systems* 27, 6, 2002.

[4] S.J. Greenspan, J. Mylopoulos, A. Borgida, Capturing More World Knowledge in the Requirements Specification. *Proc. 6th Intern. Conference on Software Engineering*, 1982.

[5] B. Henderson-Sellers and A. Bulthuis, *Object-Oriented Metamethods*, Springer, New York, 1997.

[6] B. Henderson-Sellers, "Some Problems with the UML V1.3 Metamodel", *Proc. 34th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, 2001.

[7] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, S. Eherer, ConceptBase - A Deductive Object Base for Meta Data Management. *Journal of Intelligent Information Systems* 4,2,1995.

[8] S. Joos, ADORA-*L – A modeling language for specifying software requirements (in German).*, PhD Thesis, University of Zurich, 1999.

[9] J. Ludewig, ESPRESO-A System for Process Control Software Specification. *IEEE Transactions on Software Engineering* SE-9, 1983.

[10] J. Ludewig, M. Glinz, H. Huser, G. Matheis, H. Matheis, M.F. Schmidt, SPADES - A Specification and Design System and its Graphical Interface. *Proc. 8th Intern. Conference on Software Engineering*, 1985, pp. 83-89.

[11] B. Meek, *The Static Semantics File*, ACM Sigplan Notice, Vol 25 No. 4, April 1990.

[12] K. Marriott and B. Meryer, *Visual Language Theory*, Springer, Berlin, 1998.

[13] OMG. *OMG Unified Modeling Language Specification Version 1.4*, OMG document formal/2001-09-67, 2001.

[14] F.G. Pagen, *Formal Specification of Programming Languages*, Prentice-Hall 1981.

[15] D. Ross, "Structured Analysis(SA):A Language for Communicating Ideas". *IEEE Transactions on Software Engineering*, 3,1, 1977.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs,1991.

[17] Y. Xia and C. George, "An Operational Semantics for Timed RAISE", *Proceedings of the World Congress on Formal Methods*, LNCS Vol.1709, Springer, 1999.

[18] Y. Xia and M. Glinz, *A Syntax Definition Method for Visual Specification Languages*. Technical Report, University of Zurich, 2002. http:// www.ifi.unizh.ch/req/ftp/syntax/syntax.pdf