XML SCHEMA EDITORS

A Comparison of Real-World XML Schema Visualizations

Felix Michel
Departement ITET
ETH Zürich

Erik Wilde
School of Information
UC Berkeley

December 2006
(TIK Report 265)

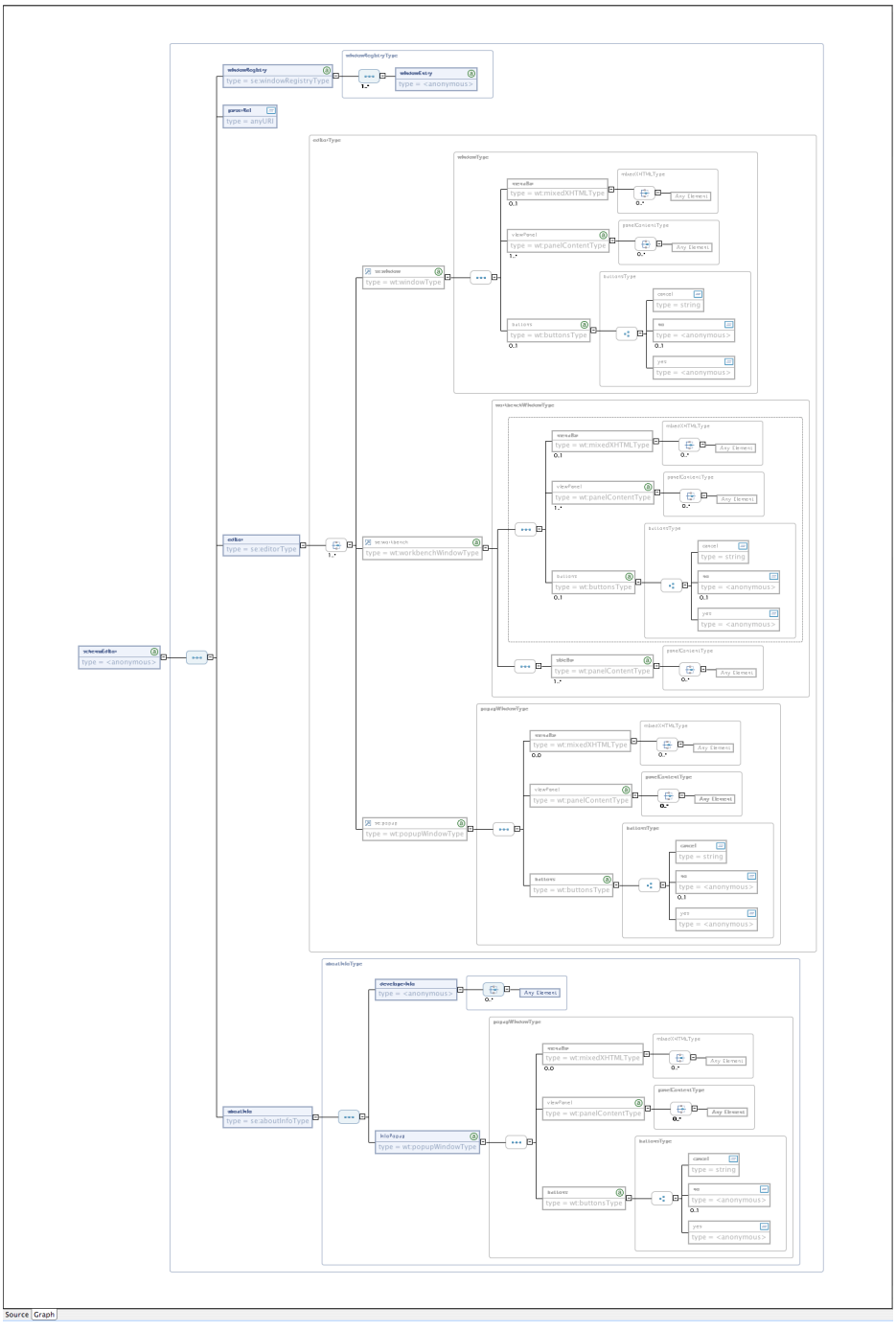


TABLE OF CONTENTS

XML SCHEMA EDITORS

A Comparison of Real-World XML Schema Visualizations

Felix Michel
Departement ITET
ETH Zürich

Erik Wilde
School of Information
UC Berkeley

December 2006
(TIK Report 265)

Abstract

XML Schema is a complex language for defining schemas for classes of XML documents, and its inherent complexity as well as its XML transfer syntax make it hard to read XML Schemas for humans. This is a problem, because in many cases XML Schemas are not only used for validation purposes, but also as the data model for classes of XML documents, which must be understood by developers working with these documents. This report looks at various visualizations of XML Schemas in existing software tools and compares them in terms of how well they are suited to represent the data model behind the XML Schema syntax. of today, no available tool provides a level of abstraction that would be appropriate for a data model perspective; most of them are visualizations of the syntax rather than the model. The tools included in this report are the Eclipse XML editors, XML Spy, oXygen, Turbo XML, and Stylus Studio. This report is not a complete evaluation of these tools, it only looks at them in terms of their schema visualization and their support of a less syntax-centered view of XML Schema.

1. Introduction

There is wide agreement that Conceptual Modeling for XML is highly desirable [wil06d]. Even though an abundance of scientific proposals currently addresses this field of research (for a comparison see [wil06d], [nec06], [moh05]), no formalism has been established so far. Yet in practice, data modeling for XML has been done for a long time, often by using schemas as a provisional model. More recent and expressive schema languages, namely XML Schema [xsd1] which has been introduced by the World Wide Web Consortium (W3C), purposely provides modeling facilities by means of its more elaborate type concepts, identity constraints, and by supporting XML namespaces [xmlns]. This increase in the schema languages' capabilities met the increasingly complex needs of a data format that, starting as a mere exchange format, eventually developed into a widely-used data format playing an essential role as a foundation to many recent technologies ([.net], [j2ee]), thus increasingly becoming part of more abstract levels of software development, where sound formalisms, methodologies and tools are indispensable prerequisites.

In the survey presented here, we investigated the tools actually available today for designing XML Schemas, directing our focus onto the particular question how *visualization* of the schema's structures was done. In our analysis, we strove to apply a perspective emphasizing a quest for *conceptual* properties throughout our comparison.

2. Goals & Objectives

2.1. Motivation

Visualization is an essential part of *conceptual modeling*. This is a recognized fact, reflected for instance by [wil06d], where *Graphical Notation* is listed as one of seven requirements for a conceptual modeling method to have.

Visualization proves helpful in different contexts of use: It enables lesser-experienced users or non-developers to use XML Schema, because ideally, it hides the difficulties of the XML syntax. Visualization is a prerequisite when discussing or negotiating the structures that are captured by XML Schema with coworkers or partners unfamiliar with XML Schema. Furthermore, visualization alleviates dealing with the data models and document grammars even for experienced users, thus fostering productivity and quality of implementation.

Contrary to other schema languages that are pure grammar languages (most notably RELAX NG [rng]), XML Schema comprises a certain amount of capacity for more conceptual data modeling itself. Especially the type concept of XML Schema allows for more advanced techniques of data modeling to be applied. However, [bex] show that these powerful features are used very rarely today. We assume the particularities of the normative XML Syntax of XML Schema to be one of the main reasons. Because we expect a good visualization to hide these particularities, we hope that visualizations as a consequence could reveal the abilities for conceptual modeling when developing Schemas, and that visualization allows for recovery of more abstract modeling properties when analyzing Schemas.

Instead of investigating or even proposing scientific formalisms for conceptual modeling for XML Schema, our idea was to look at existing visualizations in editing tools available today and at how they address the difficulties that arise from XML Schema's syntax, how they support utilization of XML Schema's sophisticated features, and to what extent they recover or reveal the underlying data model of a Schema.

2.2. Goals

This section outlines what we expect to obtain from applying our approach of looking at visualization strategies of developer tools:

- Assuming some amount of *darwinism* ruling spread and success of the tools, we hope to see more clearly what users really need and ask for when designing schemas.
- By comparing the tools' capabilities to the capabilities of scientific proposals on the one hand, and to the potential expressiveness of XML Schema itself on the other hand, we expect to be able to discern more clearly which features either of them is deficient in. Significant differences in the feature sets may hint at features worth of closer investigation. The reasons for such differences supposedly may vary, ranging from being difficult to visualize to lacking need due to very rare use of a particular feature.
- Finally, we want to:
  - develop a set of needs and requirements that can be used for profiling or even rating the tools' visualization strategies.

- offer decision support concerning which editor supposedly best fits whose needs, and in which ways future tools could be improved.

We will review those points in our conclusion.

2.3. Scope of the Comparison

We are not aiming at judging or even ranking the *editors under test*. Therefore we will not consider most of the tool's distinctive features, even though the value of a tool strongly depends on such additional features (e.g. runtime stability, performance, and capacity of improving interoperability, workflow integration, productivity, as well as possibilities for integration of external tools, etc.). Only the way of how they offer visual representations of (and interfaces to) the Schema structures shall be subject to our survey. Even the features we will look at and the criteria we will apply are chosen deliberately to a certain extent.

3. Subject of Visualization

The title of this survey promises a comparison of "XML Schema visualizations", the delicate term being "XML Schema". Even though the normative recommendation [xsd1] by the World Wide Web Consortium clearly states in section 2.2 that XML Schema is defined "in terms of an abstract data model" which "is conceptual only, and does not mandate any particular implementation or representation of this information", with the well-known XML syntax only being "a normative XML interchange format for schemas", the term *Schema* very often is used in ambiguous, inaccurate, or misleading ways.[1] Another, although minor, source of confusion is the arguably suboptimal decision to name the W3C's schema language *Schema* (with uppercase 'S'), for it is only one particular schema language out of many conceivable ones, *schema* (with lowercase 's') being the generic term or *hypernym*.[2]

Let us clarify our use of the terms first, and then declaring which one will be subject to our focus in our comparison:

- We use the term "schema" with lowercase 's' whenever a statement is including, but not limited to XML Schema, i.e., whenever we address schema languages in general.
- We use the term "Schema" when we refer to the *data model* as defined in the specification [xsd1]. We refine the term by either prepending "XML" or appending "structure" or "data model" where more explicit distinction from either other schema languages or the data representation is deemed necessary.
- The data model we are considering is the one assembled after having resolved all Schema dependencies, i.e. after evaluation of all *include*, *import*, and *redefine* statements.

---

[1] The distinction is similar to the one made for XML instances, with the *Infoset* [xis] considered to be its data model, after a vivid debate preceding today's de-facto consensus. XML Schema is lacking comparably well-established terms. This holds as well for XML's *DOM*; with *XSOM* [xso], *XSD* [wst0], and *SOM* [som] being candidates for designating the equivalent concept for Schemas.

[2] There are occasional attempts to avoid ambiguity by using the term *XSD*. But this again bears possible ambiguity, for the same term sometimes is used for stressing *data representation* (i.e., the Schema document), distinguishing it from the *data model*. To make matters worse, the Eclipse Project chose to use *XSD* as name for their Schema API [wst0].

- We use the term "Schema document" when we refer to the XML representation of the XML Schema's data model. We exclusively consider the normative XML format as defined by the World Wide Web Consortium.

Our comparison aims at inspecting visualizations of the fully assembled data model of XML Schemas as specified by the W3C.

4. Comparison Set-Up

The comparison inspects primarily static or passive properties of the editors examined, i.e. the focus is more on "how do they visualize an existing Schema" than on "how do the tools help building Schemas". The test procedure is as follows:

1. Open the principal Schema document *SchemaEditor-Schema.xsd* (see the section *Guinea Pig Schema Document* below) in the editor. Is the editor able to resolve the dependencies by itself?
2. Try to use the editor's built-in validator for Schema-validation of the Schema. As our test Schema passes IBM's SQC (Schema Quality Checker) [sqc] without problems, validation should not throw any errors.
3. Check coverage of the *Features to test* listed below.
4. Try to build the Schema sketched in section *A simple Schema to be built*, using the editor's GUI as exclusively as possible.
5. Evaluate the editor's strengths, weaknesses, and particularities by applying the *Criteria* listed below.

4.1. Features to test

As the names indicate, the *features* listed here are different from the *criteria* insofar as the former simply are the properties we will be looking for, whereas the latter are a list of guidelines used when judging the visualization's handling of the features observed.

4.1.1. Visualization

- ***Model Groups:*** What symbols / graphical conventions are used for representing the different model groups? Are there any simplifications made (e.g. nested or adjacent[3] model groups)?
- ***Mixed Content:*** Is there a graphical convention for designating mixed content?
- ***Types:*** Where / how are types visualized (if they are visualized at all)? Is there a difference between anonymous types and named types? Are *abstract* types labeled particularly? Is there a representation of (perhaps restricted) value sets? What about relations between types (i.e. inheritance; see below)?
- ***Type inheritance***: Extension, restriction. How do they visualize the fact, that inheritance is present? How do they visualize properties inherited / removed from the base type?

[3] This is easily possible as a result of inheritance by extension.

- ***Substitution Groups / xsi:type:*** How are possibility / presence of (or control over building / using, like the *block* and *final* attributes) substitution groups and type substitution represented?
- ***Constraints:*** Are any identity constraints expressible in XML Schema shown? And if so, how is this done and what details are visually discernible (e.g. element sets specified by *selector* / *field*)?
- ***Annotations:*** Where do annotations and their content show up?
- ***Invisible Properties:*** Where and how are things made accessible which have no direct graphical representation in the editor's view? This may include particle cardinality, default values for attributes, namespace information, and so on. For example, such information can be caught by using sidebars, pop up windows, or context menus.

4.1.2. "Active" Features

- ***Graphical Editing:*** Can the GUI be utilized for changing / creating Schemas / Schema documents? How does that influence the source? Are changes that has been made in the GUI reversible? Which input methods (e.g. *drag & drop*, *context-click*) can be used?
- ***Search for declaration / references:*** Built-in search for declaration of an element or type can prove very helpful. The reverse search can be even more helpful, that is, search for elements that reference global declarations or search for types that inherit a given type. Here again the question is important if the tools only consider the current Schema *document* or the whole Schema *structure* represented in the different documents.
- ***Generated documentation:*** Does the editor allow for documentation to be generated from a Schema? Does the documentation include graphical representations, and do they differ from the representations used in the editor view? Does the documentation insert additional information besides the textual information included in the Schema's *annotation* element?

4.1.3. Foundations

Is there additional information available about concepts and formalisms used in the representation? Or is the visualization even based on / using a formalism / model proposed by a third party?

4.2. Guinea Pig Schema Document

The Schema to be used in our comparison is not intended to be of real-world use. It is constructed artificially in order to cover a considerable amount of XML Schema's available features. Even though without use, we decided to build an example schema[4] that at least shows some basically meaningful properties, instead of just using element and type names like *AAA* or *Type_B*. We believe that some semantic meaning helps both to understand and to design schemas. While standard-compliance of the resulting schema alone does not give any idea whether it actually is a correct implementation / mapping / instance of an underlying conceptional model, a sensible semantic structure enables at least the human designer to perform some basic checks. Furthermore, it is much more interesting to investigate visualizations of Schemas if we can compare the visual representations against the semantic structures the Schemas are meant to be encoding.

[4] It describes a hypothetic schema editor with different windows, menus and buttons...

4.2.1. Structure

The Schema document used in this comparison is composed of different other Schema documents. There are four components (see Appendix). Two among those model the same conceptual structure by two different Schema techniques: One describes an element containing an arbitrary number of window elements of different kinds by enumerating all the respective elements in its model group; the other lists only the general window element in its model group, but builds a substitution group that comprises all the other window elements, thus allowing these other kinds of windows to be used as well. Only one of these two redundant Schema components can be used at a time, therefore an actual Schema consists of three components. Having three component documents, it is possible to employ both Schema's include and import mechanism.

All namespaces used have namespace URIs starting with the base http://people.ee.ethz.ch/~femichel/namespaces/. We therefore in the following only give the part distinguishing the namespaces, implicitly prepending the aforementioned base URI.

4.2.2. Components

Component WindowTypes-Schema.xsd defines a hierarchy of complex types that can be used for modeling different kinds of application window in an hypothetic editor. As they are considered to be generic enough to be used in any kind of editor, they are declared in an own namespace (WindowTypes-Schema). There is a base type called *windowType*, with two types inheriting from it: *workbenchWindowType* and *popupWindowType*. The former is derived by extension, the latter by restriction.

As mentioned already, the components EditorComponent-Choice-Schema.xsd and EditorComponent-Substitution-Schema.xsd model the element containing the different windows by means of a choice model group and substitution group respectively.

Component SchemaEditor-Schema.xsd finally is the central Schema including and importing the components described above – a kind of main entry point, if we employ a term known from Java. Although other global elements can be found in referenced components, *schemaEditor* is the single globally declared element within this component's document. This resembles to the well-known venetian blinds pattern and gives at least a hint on what the root element of the instances described by the Schema is supposed to be.[5]

4.2.3. Features

The Guinea Pig Schema covers the following of XML Schema's features:

1. *Choice*, *sequence*, and *all* model groups
2. Local and global definition of elements and types (the latter as named or as anonymous types)
3. Simple types, restricted simple types by *enumeration*, *range constraining facets*, and by *regular expression patterns*
4. Complex types with complex and simple content
5. *Mixed* content

[5] The fact that XML Schema has no syntactical mechanism to unambiguously designate the instance's root element is a particularity often criticized.

6. Complex type inheritance by *restriction* and by *extension*
7. Schema constraints: *Unique*, *Key*, *Key* with *Keyref*, circular constraints[6]
8. Wildcard elements (narrowing the set of possible elements by use of the *namespace* attribute)
9. Use of multiple (also third-party, e.g. XHTML, SVG) *namespaces*
10. *Substitution groups*
11. *Include* and *import* of external Schemas
12. Documentary annotations, using the xml:lang attribute and the ISO 639/3166 language encoding rules.

4.3. A simple Schema to be built

A simple Schema shall be developed using the generic window types from WindowTypes-Schema.xsd. The target namespace is SimpleEditor-Schema, and the target structure looks as follows:[7]

```
<pseudoSchema
     targetNamespace="http://people.../SimpleEditor-Schema"
     xmlns:ed="http://people.../SimpleEditor-Schema"
     xmlns:wt="http://people.../namespaces/WindowTypes-Schema"
     xmlns:xs="http://www.w3.org/2001/XMLSchema">

     <!-- A simple editor -->

     <type name="simpleWindowType" extends="wt:windowType">
          <sequence>
          </sequence>
          <attribute name="isSimple" type="xs:boolean" default="true"
     </type>

     <element name="simpleEditor">
          <sequence>
               <element name="registry">
                    <sequence>
                         <element name="registryItem" multiplicity="[1…∞]">
                              <attribute name="windowRef"
                                   type="xs:integer" />
                              xs:string
                         </element>
                    </sequence>
               </element>
               <element name="editor">
                    <choice>
                         <element name="window" multiplicity="[1…∞]"
                              type="ed:simpleWindowType">
                         </element>
                    </choice>
```

[6] This isn't a pathological case at all: Here it is used for a 1-to-1-mapping between windows and registry entries.

[7] Because so many differently flavored schema languages exist today, but still no established formalism on a conceptual level, we dare using yet another (*pseudo-code*) formalism to specify our schema structure.

```
            </element>
        </sequence>
        <constraint
            id="ed:editor/e:window/@windowID"
            idref="ed:registry/ed:registryItem/@windowRef" />
    </element>
</pseudoSchema>
```

Actually, it would have been far more interesting investigating an inheritance by *restriction*, because restriction in XML Schema is a rather troublesome thing, potentially introducing even erroneous modeling, thus being a much better example for where tool support could ease Schema implementation quite reasonably. But as apparently no tool examined (except for Altova's *XML Spy*, see below) offered support for complex type restriction, we decided to alleviate the requirements.

4.4. Criteria

In this section, we give a brief description of our criteria in list form. We tried to discern more or less independent[8] dimensions of criteria where this was possible and appropriate.

1. **Coverage of the *Features to Test*** (as listed above): Which of the features are available at all, and which of the problems that arise when visualizing these features are addressed at all?
2. **Perspective / Focus**: A visualization has to choose a certain perspective, stressing a particular focus (although such perspectives and focuses can be implicit, unconscious, and therefore possibly inconsistent, overlapping, and unclear).
   - 2.1. **What**? On which entities the focus is directed? In the particular case of XML Schema, we can think of these perspectives:
     - 2.1.1. Element-centric
     - 2.1.2. Type-centric
   - 2.2. **How**? Different ways of representing the relationships among those entities are conceivable, namely:
     - 2.2.1. Tree structure (hierarchical)
     - 2.2.2. Global enumeration (flat)
     - 2.2.3. A (possibly circular) graph
   - 2.3. As perspectives substantially influence the users' understanding of the structure represented, and as different perspectives might be appropriate in different contexts, it perhaps would be desirable to have **multiple** or **selectable views**.
3. **Subject** of the visualization: In section *Subject of* Visualization, we pointed out the subtle, but essential differences of data model, assembled data model, and data representations of schemas. We also stated what we are requiring to be the subject of visualization: the assembled data model. In practice, the editors under test might focus on:

[8] Mathematically spoken: "perpendicular"

3.1. The schema *document(s)* (the data representation)

3.2. The schema (the data model)

3.3. The assembled schema (the data model, with all dependencies resolved)

3.4. A conceptual model describing the data structures captured by the Schema

4. **Purpose** of the editor: Schema editors may have different purposes, resulting in different emphases, favoring different perspectives, etc. We first distinguish whether the visualization primarily is intended to be used as an *input* mechanism, or rather as a *read-only* depiction. (Of course, the editors inspected here are all of the former kind, but most of them include mechanisms for *documentation* as well.)

   4.1. Design View/ GUI (**active**): The distinctions we make here obviously are strongly related to the criterium 3. Is the tool intended to be a:

      4.1.1. Is it a **document editor**?

      4.1.2. Is it a **data modeling tool**?

   4.2. Visualization (**read-only**): These distinctions are related to two key benefits a conceptual modeling view would provide. Is the visualization meant to be used for:

      4.2.1. **Documentation** (primarily addressing developers)?

      4.2.2. **Specification** / **Negotiation** (addressing non-developers as well)?

4.5. Editors under Test

From many editors available today[9] — either as standalone applications or as parts of *IDEs* (Integrated Developing Environment) — we chose the following:

1. The built-in Schema editor from Eclipse 3.1.2 Webtools' WST (Web Standard Toolkit), version 1.0.3
2. The built-in Schema editor from Eclipse 3.2 Webtools' WST (Web Standard Toolkit), version 1.5.0
3. Altova's *XML Spy*
4. SyncRO's *oXygen* (as a standalone application[10])
5. Tibco's *Turbo XML*
6. DataDirect's *Stylus Studio* (Enterprise Edition)

The first two editors are chosen as representatives of *Open Source* software. Furthermore, the WST Schema editor has recently undergone a fundamental redesign, now having a unique *type oriented* perspective. We considered this change, and the resulting new editor, a fact particularly worthy of investigation.

---

9 For a list of tools, see [cov1] and [xsd4].

10 The editor is also available as a plugin for the *Eclipse* IDE.

To our knowledge, number 3, 4, and 6 are the XML Schema editors most commonly used among XML developers. Number 5 has been included in our comparison because it comprises interesting concepts of visualization somewhat different to the three above.[11]

4.6. Difficulties

A few practical problems have influenced our comparison:

- The high configurability of some of the tools made it very hard to compare the tools. As a rule of thumb, we primarily based our evaluation on the default settings and mentioned further possibilities for configuration in the textual description and in the tabular comparison.
- In some cases, problems occurred that very likely may have been problems of our particular set-up of the respective tool rather than being general problems of the tools itself. We always indicated such problems when we were aware of them.
- Even though this comparison has been done with due diligence, it still is possible that we missed some of the numerous features of the more powerful tools.

5. The Comparison

Before presenting the outcomes of our comparison in a tabular form, a textual description presents the editors and their particularities in more detail. Features that are part of the list stated above but are not described in this more descriptive section can be found in the tabular view.

5.1. Textual Description

5.1.1. WST 1.0.3

WST 1.0.3 has a clearly *element centric* perspective, showing the elements' and attributes' types as "type=*typename*" (where *typename* is the actual type name) in boxes beneath the element names. For elements that have anonymous types, the boxes contains "type=<anonymous>", explicitly indicating anonymous type.

The nesting of elements is mapped to a nesting of boxes that surround model groups. The picture selected for the title page of this comparison shows the Guinea Pig Schema as it is rendered in WST 1.0.3. Completely unfolded, the visualization shows the nested structure of model groups.

External includes and imports are resolved in the graphical view (but the sidebar containing the list of elements, types, attributes etc. does only comprise document-local entities). Elements and types defined in external documents are displayed grayed out and are read-only. Clicking those corresponding boxes doesn't open the documents in a new editor; one has to open the respective documents explicitly. At least all information usually displayed in sidebars is shown for those external entities as well.

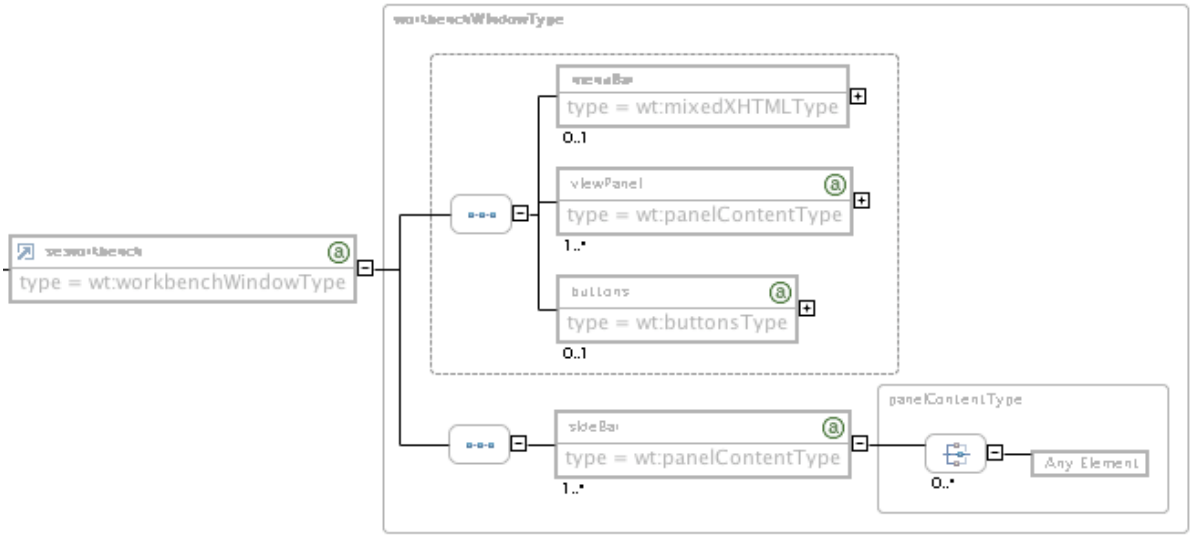
[11] The *Turbo XML* editor also is part of the comparison in [moh05].

The following symbols are used for the different *model groups*:

Choice *Sequence* *All*

Cardinalities of occurrence are written beneath the symbols in the form *[1…∞]*. This holds as well for elements and model groups.

Next, we inspect the approach WST 1.0.3 takes when visualizing type relationships, i.e. type derivation.[12] Type inheritance by *extension* is visualized as follows:



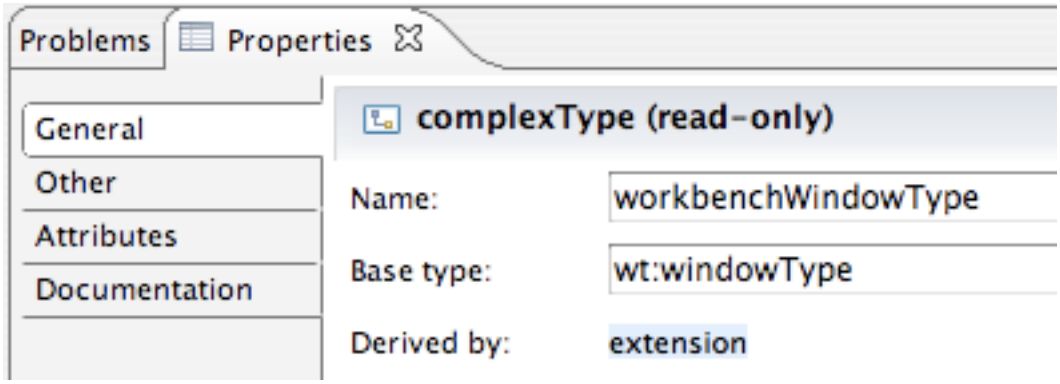
Type inheritance by extension in WST 1.0.3: The model group inherited form the base type is displayed in a dashed box.

Attributes are listed in a sidebar, elements containing attributes are showing a green "a" symbol in their top right corner. Note that only the elements rather than the types are decorated this way; in fact, the type *workbenchWindowType*, which is an externally declared global named type, gives no clue that it contains attributes as well, when it is displayed standalone.

Furthermore, attributes inherited from a base type are not listed anymore for types derived by extension.

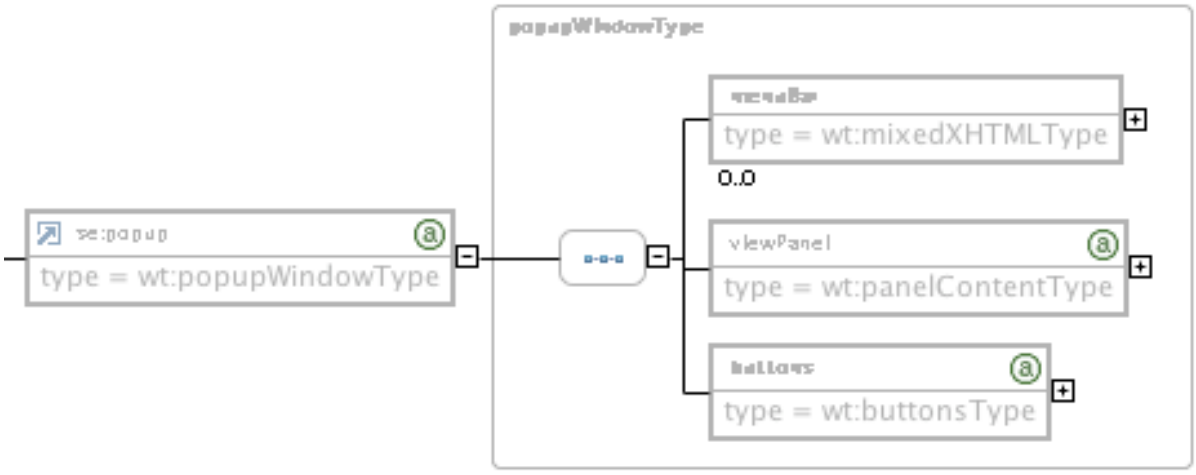
The base type is not determinable other than by looking at a sidebar:

[12] We use both terms, type *derivation* and *inheritance*. Even though from a purely theoretical point of view, these concepts are carefully to be discerned, we might claim our sloppiness to be rooted in Schema's similar attitude.



Sidebar indicating the base type of the derived type workbenchWindowType *in WST 10.0.3.*

This is the visualization of inheritance by restriction:



Type restriction in WST 1.0.3: Elements restricted not to appear anymore are still displayed (menuBar).

The element *menuBar* is optional in the base type's declaration and is restricted away in the derived type. The tool still displays the element, specifying a cardinality of occurrence of zero. This is correct with respect to the XML syntax, yet the resulting grammar does not allow this element to appear in the derived type's model group. Attributes that are subject to restrictions (and therefore to be repeated in the Schema document) are visible in the sidebar mentioned above, even though showing none of the restrictions applied.

Building the simple Schema:

In our setup, using WST 1.0.3 as a graphical editing interface was complicated through severe rendering problems in the attribute pane. It is therefore possible that we may have overlooked some features.

The principal input method is context-click. As there is no single element representing the content model, one directly can append model groups to the elements. This speeds up building the structure significantly and hides some of Schema's syntactical complexity from to user. On the other hand, it apparently makes it impossible to create complex types with simple content. Setting the elements "type" property to a simple type doesn't introduce a simple content model, but makes the entire element a simple type, removing all attributes defined before.

Declaring identity constraints is not possible in the GUI at all.

A nice feature WST 1.0.3 shares with its successor discussed below is that on creation of new Schema documents, it automatically adds a *targetNamespace* attribute containing a default namespace, and it adds a namespace declaration for this namespace at the same time. This again supports the user in creating valid and functional Schemas, circumventing some of the syntax's pitfalls.

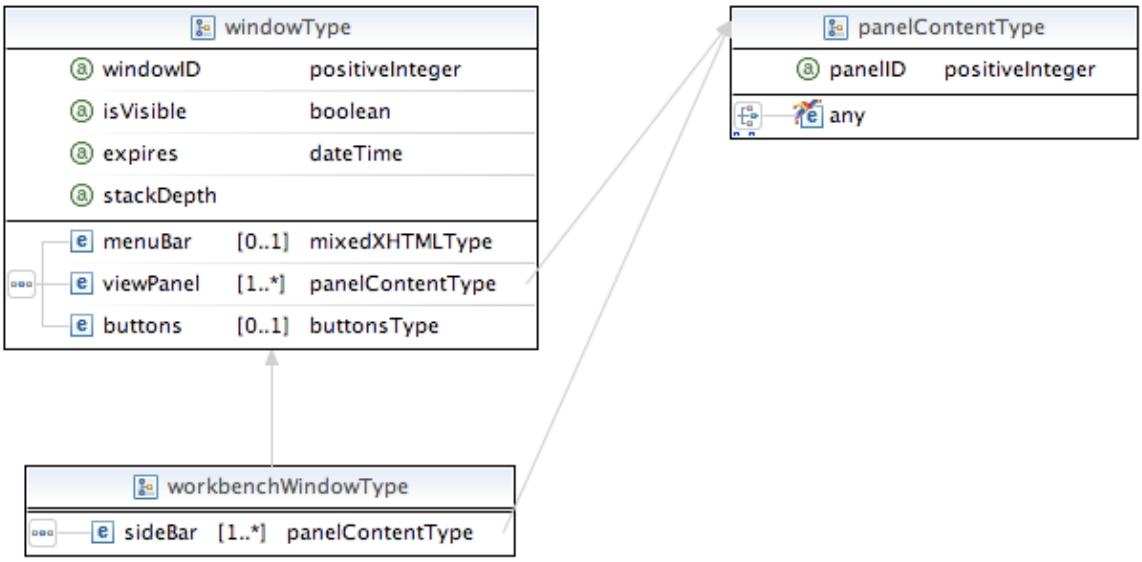
5.1.2. WST 1.5.0

Compared to the prior version, the authors of the WST Schema editor decided to change the concept of visualization quite radically. Surprisingly, this fact has not provoked any reactions in Eclipse's webtools newsgroup.[13] Our interpretation is that either the superiority of the new version puts aside any comments, or that only very few developers are using the GUI at all. Anyway, the Schema editor in WST 1.5.0 pursues a unique *type-centric* perspective.

Additionally, no more than two hierarchical tiers get displayed at a time, resulting in a almost flat depiction of entities. While this may enforce a neat overview and preserve from having the screen filled with deep hierarchies unfolded, it does not shield users from getting lost in such hierarchies, and it can be very cumbersome to deal with in practice.[14]

For the model groups within those hierarchical steps displayed in the first place, the same symbolism as in WST 1.0.3 is used, except for slightly updated coloring.

The type-oriented perspective obviously unveils its strength when representing the relationships between types, i.e. *type derivation*. Derivation by *extension* is visualized as follows:



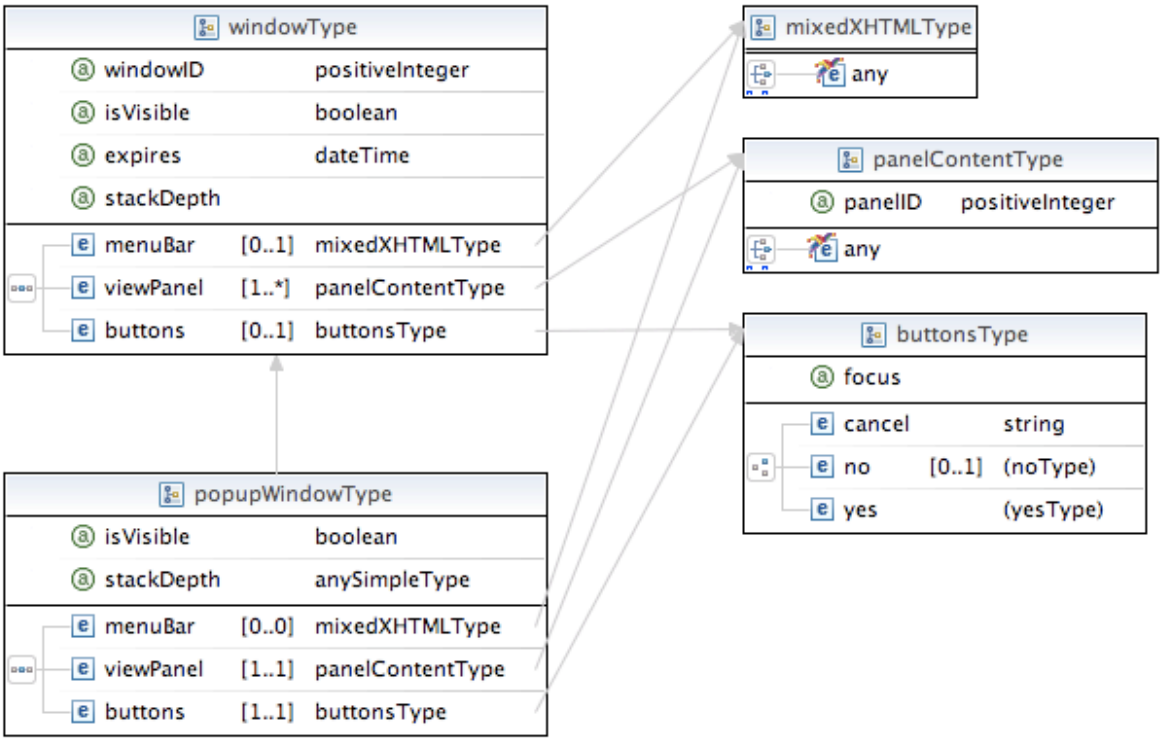
Complex type extension in WST 1.5.0. Notice that only the direction of the arrows discerns type assignment

[13] news.eclipse.org/eclipse.webtools; see Keith Chong's entry from April 7, 2006

[14] As such restricted display of hierarchical structures is not an inherent property of type-oriented perspectives however, we presume that rather implementational reasons caused this peculiarity.

from inheritance relationship. Furthermore, one can see that the attribute stackDepth*, which is of a restricted type, does only show its type information in an external sidebar.*

Whereas derivation by *restriction* looks as follows:



Complex type restriction in WST 1.5.0. When comparing to the figure above, it can be noticed that there is no difference in arrow style in order to distinguish between inheritance by restriction / extension. Also, the complete repetition of the base type's content model (perhaps an unfortunate concept in XML Schema itself) is directly visualized, obscuring inheritance substantially.

Although derived types graphically show the connection to their respective base type, there is no concept representing the reverse: A type used as a base type does not show types that are inheriting from it (nor does it show elements that refer to it). Furthermore, there is no graphical distinction made between the different kinds[15] of derivation. And again, the visualization of the derived types is much more a visualization of how the derivation is expressed in the syntax of the Schema *document*, rather than a representation in terms of the data model or even on a more conceptual level. For instance, in the example above, the attribute *stackDepth* is shown, whereas the attributes *isVisible* and *expires* are not. The reason is that the former has to be repeated in the document's syntax (because it is restricted!), while the latter are simply inherited. This is a prototype case where representing the Schema document substantially obscures the meaning of the data model: The former attribute in fact is *not* part of the derived type anymore, but the latter still are. The visualization shows confusing information.

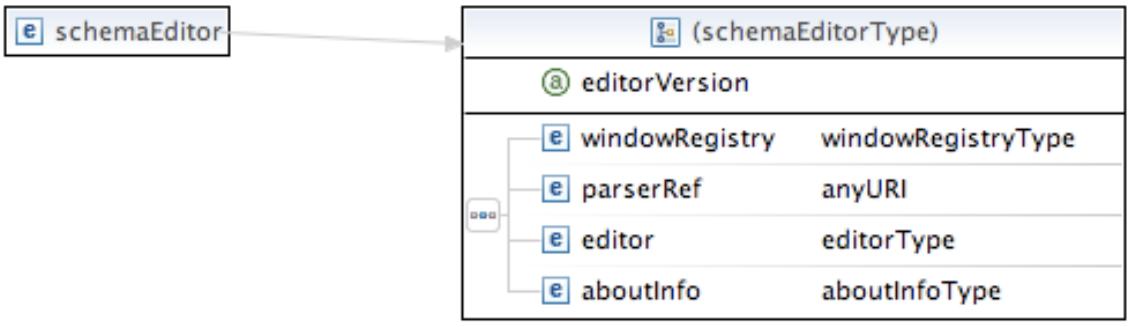
The tool successfully resolves all references; elements, types, etc. included or imported are shown grayed out.

[15] For clearness, we will use the term "kind" rather than "type" when distinguishing the different mechanisms Schema provides for derivation, as *type* is already used for the *types* involved in such a derivation.

WST 1.5.0 shows anonymous type definitions in boxes like it does for named types, substituting an auto-generated name built by concatenating the parent element's name with "Type", displayed in parentheses.

The following anonymous type is displayed as follows:

```
<element name="schemaEditor">
      <complexType>
            <sequence>
                  <element name="windowRegistry"
                        type="se:windowRegistryType">
                  </element>
                  <element name="parserRef" type="anyURI"></element>
                  <element name="editor" type="se:editorType"></element>
                  <element name="aboutInfo" type="se:aboutInfoType"></element>
            </sequence>
      </complexType>
      ...
</element>
```



*Anonymous types in WST 1.5.0. Notice that the model group and any attribute are attached to the **type** (right box) rather than to **elements** (left box).*

As in WST 1.0.3, identity constraints can neither be displayed nor generated.

The "up" button navigates based on *documents* rather than on the *Schema structure*: When navigating upwards within a document included or imported, one ends up on the top level of this document rather than on the top level of the assembled Schema's root level.

An other fact indicating that many features are addressing the Schema *document* rather than the Schema's data model: In part two (*"Active" Features*) of our list of *Features to test*, we claimed it to be desirable to have a tool listing all references of a given type or element globally declared. Such a feature indeed is part of WST 1.5.0, but it manages only to list references in the current document rather than in the current *Schema*.

Building the simple Schema:

The graphical editing capabilities are incomplete (e.g., we could not set a default value for attributes) and partly erroneous (e.g., removing the extended type's empty sequence removes the base type's model group in the visualization). Probably both facts are due to the recency of the tool. Upcoming versions might fix these problems.

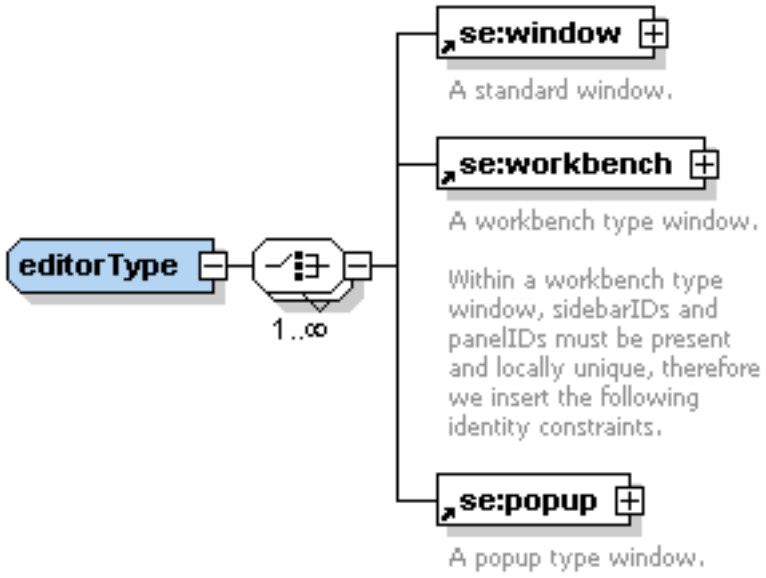
Exactly as in WST 1.0.3, one cannot define complex types with simple content.

There is no possibility to *undo* operations in the graph view; but changes done in graph view can be undone in the source view.

There is a pattern editor for building regular expressions used for simple type restrictions.

5.1.3. XML Spy

Altova's XML Spy's strategy when visualizing Schemas is a element-centric one. The symbols representing the model groups are quite similar than the ones found in WST, but model groups with non-single cardinality are depicted in a manner of stacked symbols. The following *choice* group shows this:

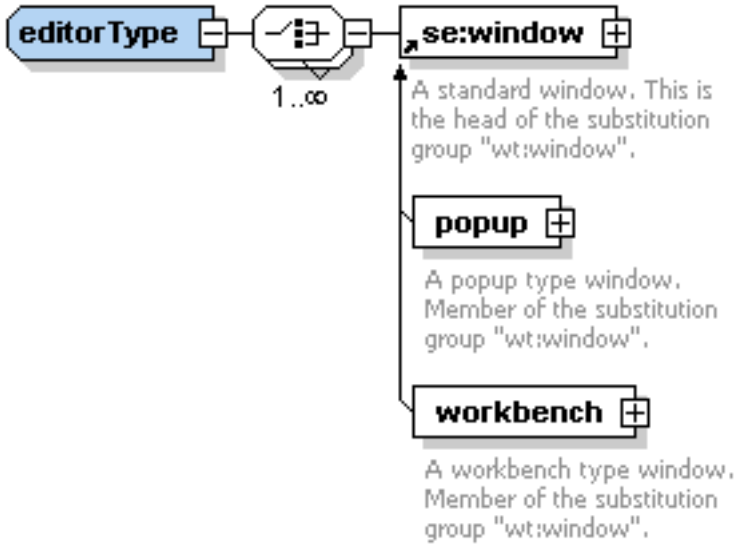


A choice *group in XML Spy. The model group symbol is annotated with the cardinality. Documentary annotations are included into the graphical view. The little arrows prepending elements' names indicate that these elements are referencing globally declared elements.*

The content of documentary annotations – if present – is displayed in proximity of the elements it describes. This is a much more *conceptual* way to do it, than to emphasize the annotation *elements* and their place within the Schema document's structure.

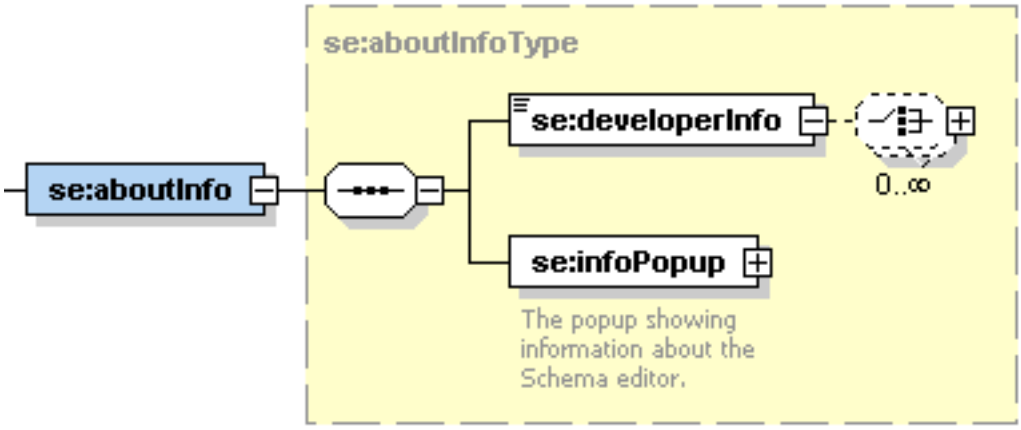
XML Spy is one of the few editors that provide an appropriate formalism to represent content models using *substitution groups*: Most often, substitution groups are used by Schema designers as a functional equivalent to *choice* groups (however with a possibly different semantic connotation). We consider it therefore to be appropriate to visualize *appearances* of members of a substitution groups in a similar way as the members of a choice model groups would be visualized.

The appearance of a model group containing a member of a substitution group is rendered as follows:



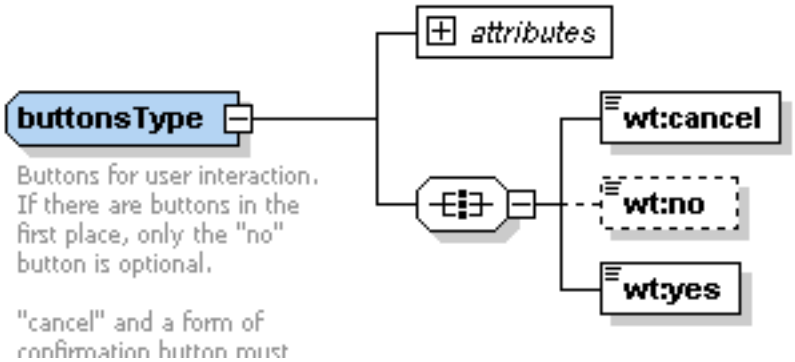
Visualization of a substitution group *in XML Spy. We chose to display this case amidst the model groups in order to stress the visual similarity to the* choice *group (see above) achieved in XML Spy.*

The *sequence* model group is rendered by XML Spy this way:



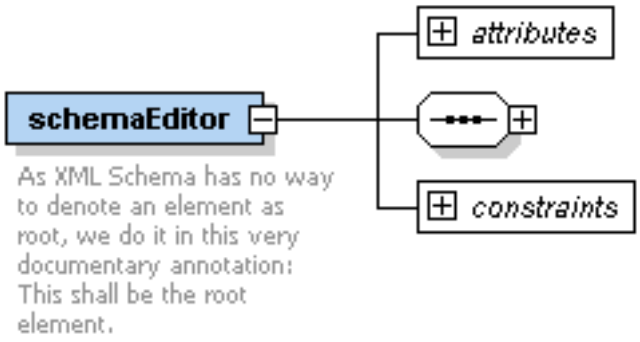
A sequence *in XML Spy. While the parent node of the tree visualization in the example above was a named type, here* aboutInfo *is an element, the named type referenced is shown in a yellow box. Note the representation of* mixed content *in* developerInfo*: A little symbol in the top left corner indicates character content, the model group attached to the right complex content.*

From the example above we can see that optional model groups are indicated by dashed surrounding boxes. The same holds for elements, as the next example, which shows the *all* model group, demonstrates:



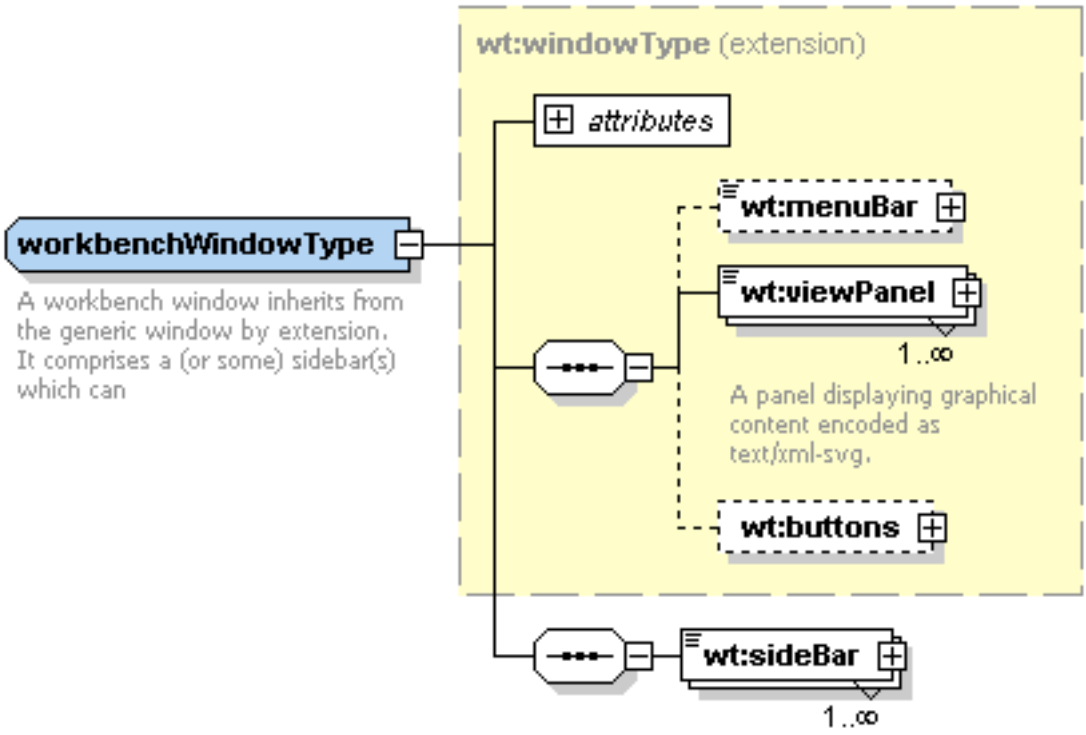
The all *model group in XML Spy. Note that optional elements are indicated by a dashed box attached by a dashed edge.*

Attributes are displayed as a dedicated child element of the parent element (if in the context of an element – regardless whether its type is local anonymous or a referenced global type) or the parent global type (if in the context of a global type definition). Constraints are shown in a similar way, except that their box is rendered on the bottom branch originating from the same fork. Note that using this notation the visualization tree obtains an additional, intermediate forking which represents the fact that attributes and constraints are part of the element declaration, but not of the model group. While the two outer branches connect to the attributes and constraints, the middle branch forks into the subtree that represents the model group:



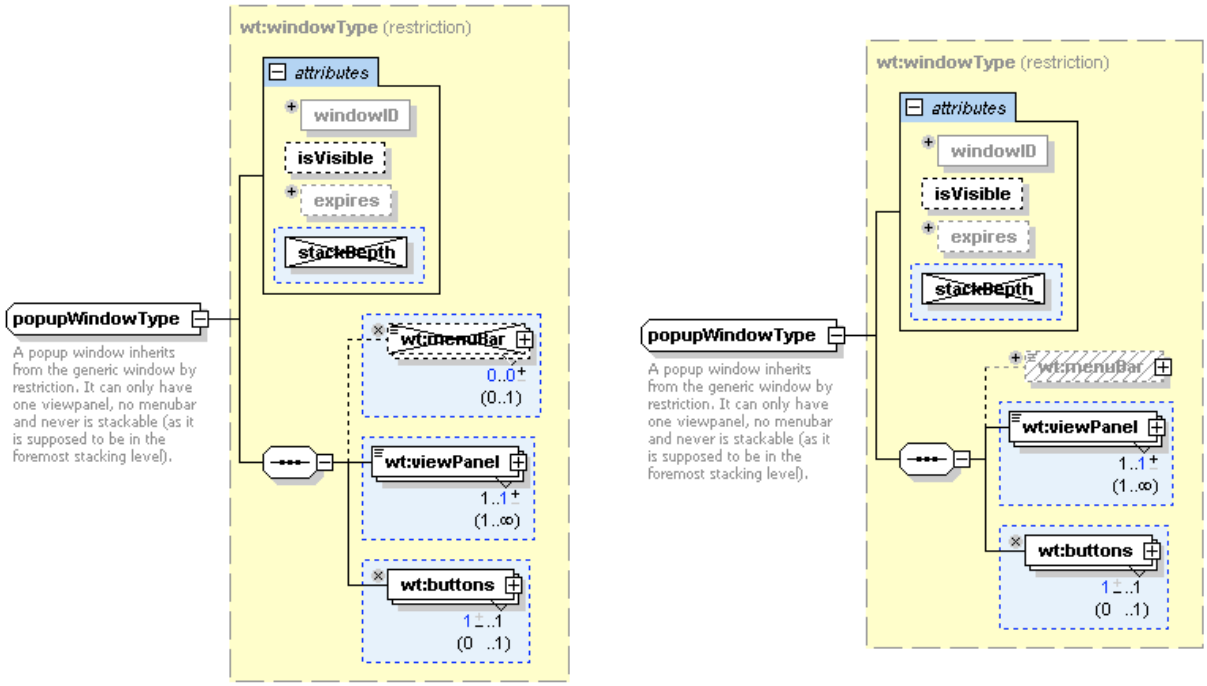
The basic visualization of an element of complex type in XML Spy: There are three basic branches originating from the element's collapse symbol: attributes, model group, constraints.

Although element-centric, XML Spy provides elaborate visualization of type derivation. The connection from derived to base types has no graphical representation, but at least the base types name and content model is displayed indicating the kind of inheritance as well (both base type name and kind of inheritance in a textual way in the top left corner of the box surrounding the base type's content model). The content model inherited attached to the first of the two trees discussed before, backed by a yellow panel clearly distinguishing it from local definitions:



Type inheritance by extension *as represented in XML Spy. Base type and kind of inheritance are enumerated within the yellow box showing the base type's content model. The derived type's model group is appended to the main branch.*

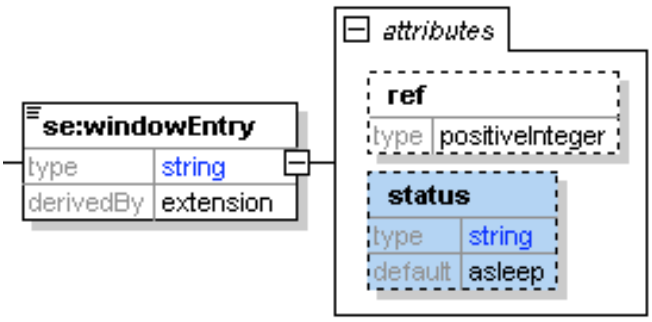
Compared to type *extension*, we consider *restriction* to be more complicated for the aforementioned reasons, i.e., because of the particularities of the XML syntax of XML Schema: *Attributes* require to be repeated in the derived type's declaration, if they are to be restricted. Most often, this restriction simply is the prohibition of attributes that were optional in the base type. Parts of the *content model* are required to be repeated, if they are to be inherited. The problems arise form these two aspects: The former necessitates inspection of the base type (for the derived type's data model usually cannot be inferred from Schema's XML syntax), the latter makes it easy for a developer to write incorrect (in terms of his intentions) or invalid (in terms of Schema's XML syntax) code. We expect a visualization to address both points. XML Spy attempts to do so:



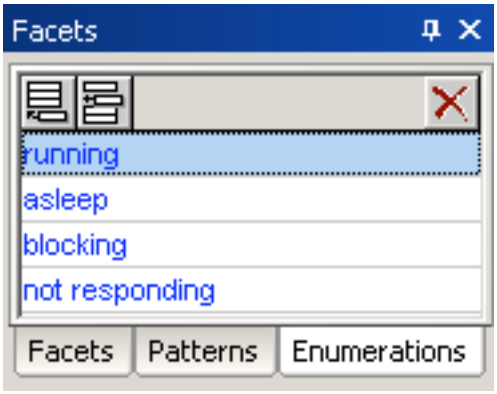
Type inheritance by restriction *in XML Spy. The inherited content model as well as restrictions applied are represented correctly. The left figure shows the editor recognizing that* menuBar *has trivial cardinality, offering a close box next to it. The figure in the right column shows the rendering after having completely removed* menuBar *from the derived type's content model.*

Even though type restrictions are not shown to that extent when displayed in a bigger context (e.g., when showing subtrees where some elements are using this type) for the sake of visual clarity, this addresses the intrinsic problems of the syntax of type restrictions: Attributes inherited from the base type, but not literally repeated in the XML syntax are shown as well, but greyed-out; non-optional parts of the content model are outlined in red, if omitted illegally. Furthermore, every box representing parts of the base type shows a little symbol indicating the operations permitted for restriction: Parts that are optional in the base type for example show a close box, indicating that such a part legally may be omitted in the derived type. Finally, parts of the base type whose use have been prohibited in the derived type are displayed stroke through.

Attributes show only their name; using a button called "Add predefined details", type and default values are added in the graphical view. Further details like constraining facets are displayed in a special pane.

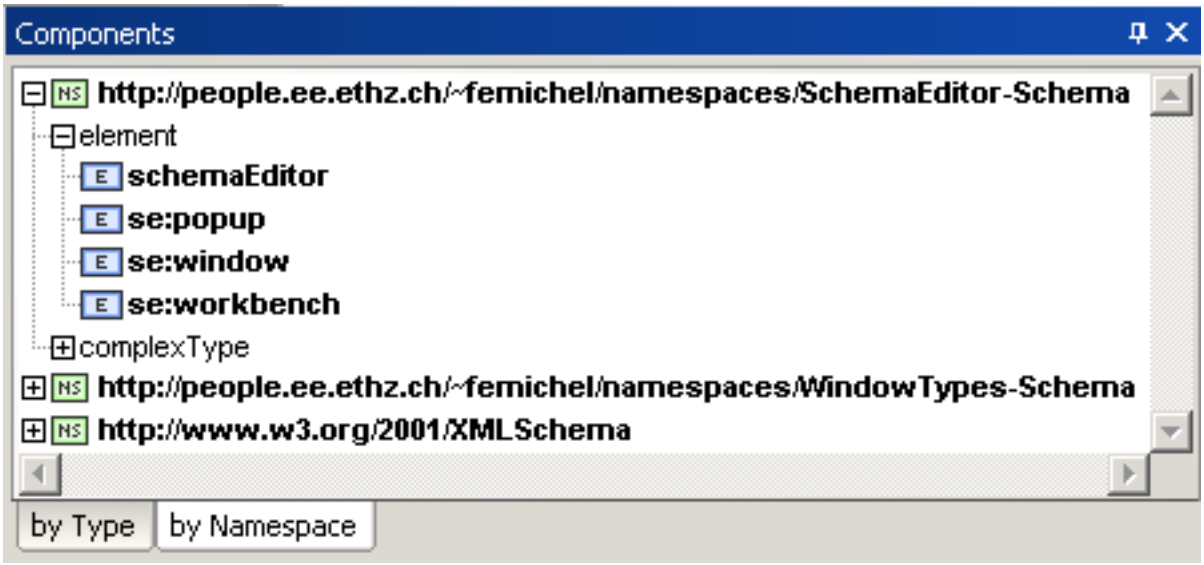


A complex type with simple content in XML Spy. Note that elements and attributes show more detailed information than in the examples above; this is selectable via a button called "Add predefined details".



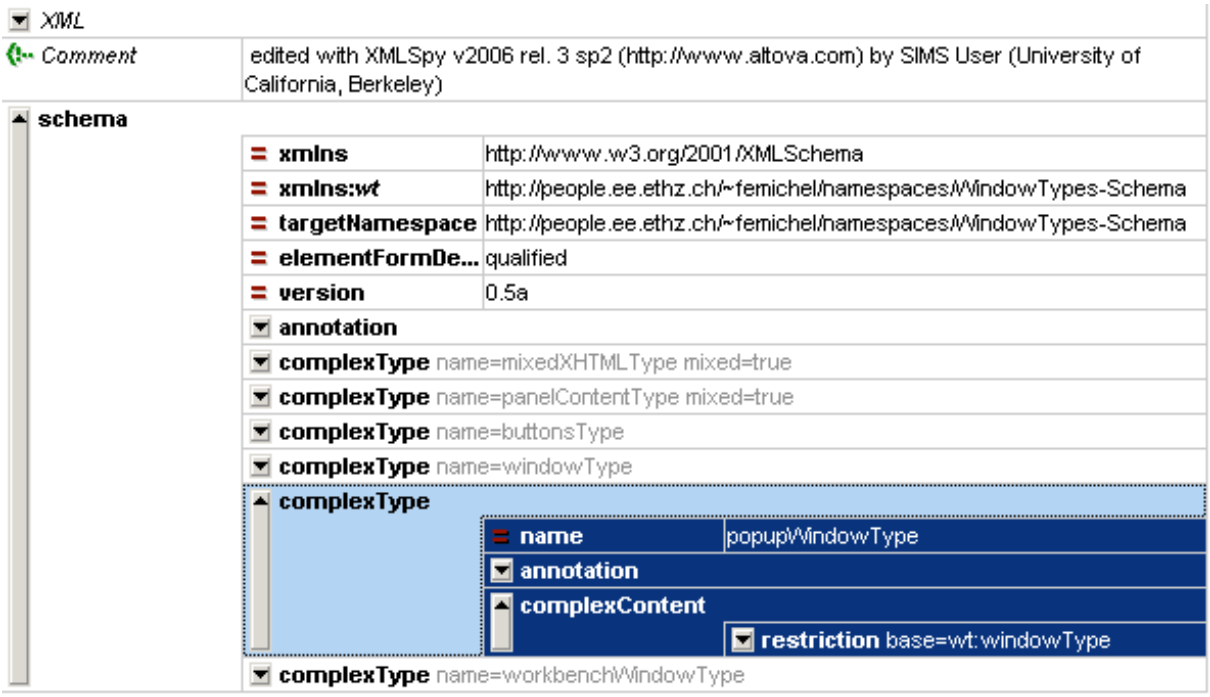
The value set of the attribute status, *which is restricted by enumeration, can be seen and edited in a special Facet pane.*

As in most of editors inspected, there is a list view of elements and global types. XML Spy offers an additional view of that list ordered by namespaces.



The Components *sidebar in XML Spy. The list of global elements and types is available in a view sorted by namespaces as well, including the built-in types from W3C Schema namespace.*

Besides the graphical view discussed, XML Spy also provides a so called "Grid View". This is a slightly varied form of a tree view and is much closer to the XML syntax of the Schema document. Thus, it reveals much more detail and offers more control for experienced users. The purpose of this view is clearly more being a *document editor*, and it can be used for editing instance documents as well.



The Grid View *in XML Spy. This is a view of the Schema document tree.*

Building the simple Schema:

The *import* has to be done either in a list view that "displays all globals" or in the grid view. After doing so, an appropriate namespace declaration is inserted automatically.

Deleting global elements or types seems to be impossible in the purely graphical view; one again has to go to the aforementioned list view.

In the graphical view, cardinalities can't be specified exactly; only "optional" and "unbounded" are offered.

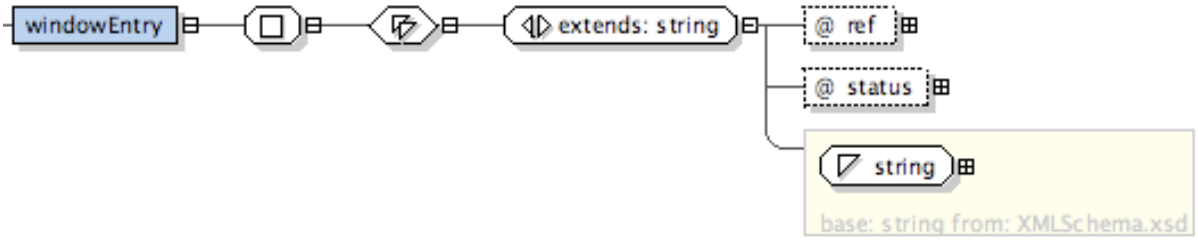
Derivation has to be started in a sidebar, but once the derivation is defined, its editing is heavily facilitated by the features mentioned.

XML Spy rewrites files opened. Most prominently, it inserts a comment line on the documents top line. Furthermore, it removes unnecessary facets like {*min, max*}*Occurs="1"*.

5.1.4. oXygen

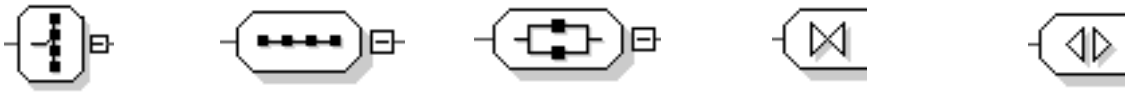
SyncRO's oXygen editor clearly is element-centric and focuses on the Schema document much more than on the data model. The tool's symbolism even is mostly a one-to-one representation of the Schema's document structure. This basically means replacing nested pointy brackets by boxes connected by a tree[16]. This at least has the advantage that Schema features often disregarded in other visualizations are shown in oXygen: For example, complex types containing only simple content are visualized as follows:

[16] An indication of this approach is the fact that the Schema document's root element *xs:schema* is the root element of the representation – even though this element has nothing to do with the conceptual structure *represented* by the Schema document.



Simple content in a complex type in oXygen: The boxes stand for (from left to right): The element using the complex type, the complex type, the simple content, the relation (extension) to the base type (xs:string, in a yellow box as the type is defined externally) and finally the attributes making necessary introduction of a complex type in the first place.

In more detail, the symbols for the model groups and the inheritance kinds:

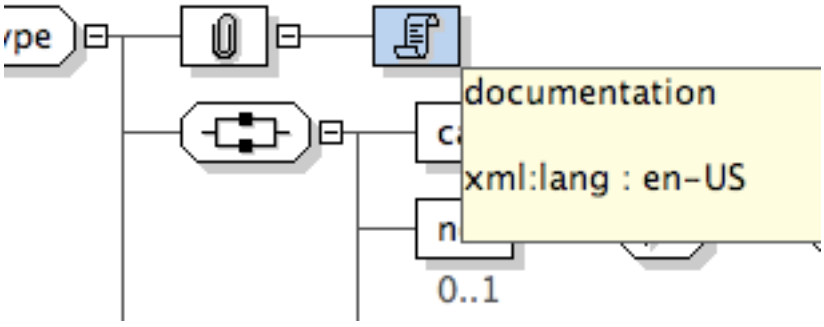


Choice *Sequence* *All* *Restriction* *Extension*

The aforementioned differentiation of {complex, simple} × {content, type} is built from these symbols:

Simple type *Complex type* *Simple content* *Complex content*

While this degree of detail can be useful in some cases (especially for the developer having in mind the structure of the underlying Schema *document*), it obscures and confuses the visualization of the Schema's conceptual structure in most cases. Probably the most obvious example is the representation of the *annotation* element:



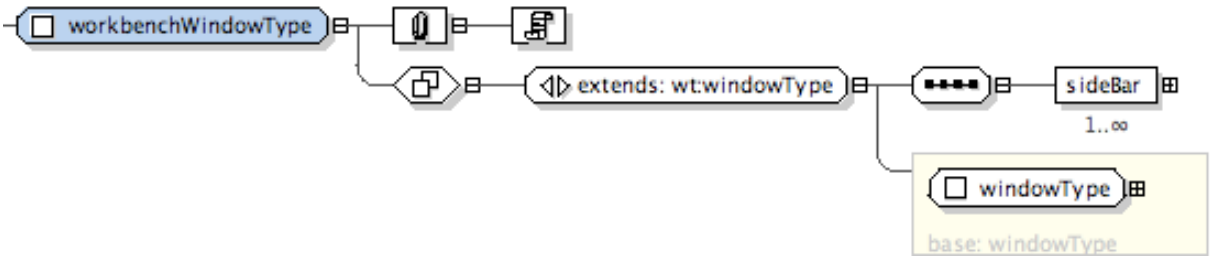
Rendering of annotation *elements as nodes in the tree structure.*

Showing those elements as tree-nodes and -leaves perhaps is consistent with the Schema document (i.e., the Schema's *serialization*), but can be considered quite confusing from the conceptual point of view.

Using a graphical formalism very close to the XML syntax, visualization of type derivations becomes rather verbose and lengthy. oXygen uses dedicated symbols representing derivation, the derived type's content model is attached to that symbol as well as the definition of the base type. This again obscures the structural relationship as it exists from a data

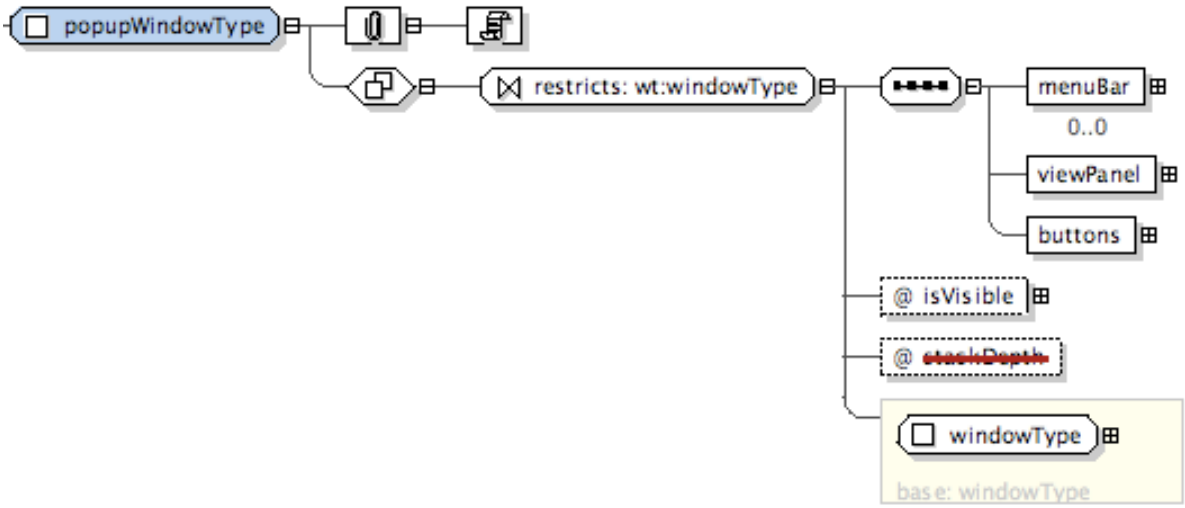
model perspective: Rather than merely being *somehow attached* to the derived type, the content model (i.e., the attributes and model group) is first inherited and then extended or restricted. In the formalism chosen here, this relationship does not have a appropriate graphical expression.

Type derivation by *extension* looks like this:



Type extension in oXygen: The derived type's model group is drawn in the usual way model groups are; the inherited content model is attached before the model group box.

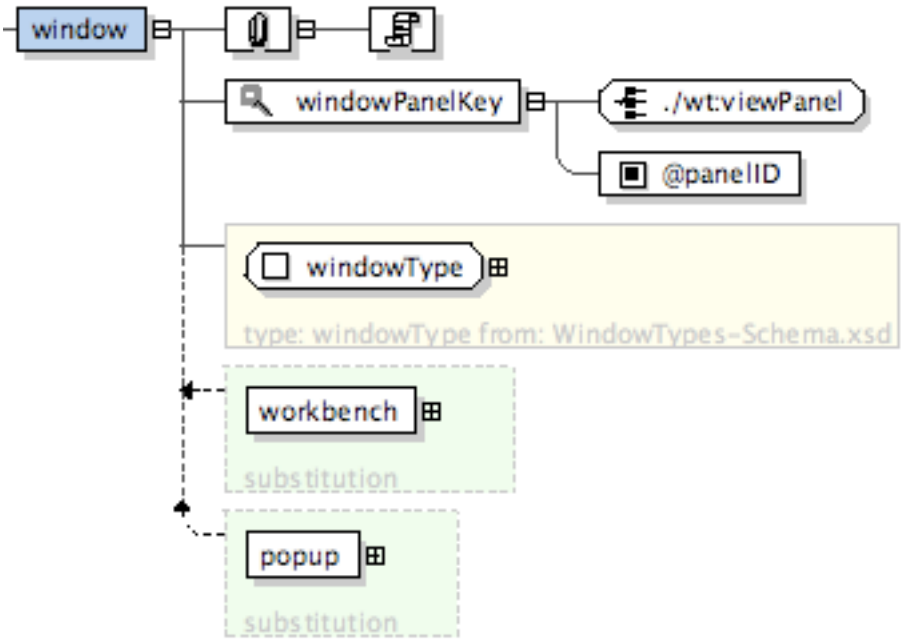
Only a developer familiar with Schema's XML syntax recognizes that in an XML instance, an element *sideBar* will be treated equivalently to all other elements of the inherited model group. The visualization does not reflect this. Derivation by *restriction* is rendered as follows:



Type restriction in oXygen: The fact that the attribute stackDepth *is stroke through does not reflect a consequence of the restriction; it is just the way oXygen renders prohibited use of attributes.*

Referring to the thoughts on restriction discussed in the section above, we clearly see the two problematic properties of Schema's syntax translated to, but not resolved by a graphical representation. Again, properties *not* present in the data model are shown (the attribute "stackDepth", because it has to be repeated in the XML syntax), whereas properties present *indeed* are omitted (the required attribute "windowID" inherited from the base type).

Substitution groups are recognized and displayed. Dashed lines and little upward arrows connect green boxes containing the elements participating in the substitution group to the line between the substitution group head's element and type symbol:



A substitution group as rendered by oXygen. window *is the head,* workbench *and* popup *are members of the substitution group.*

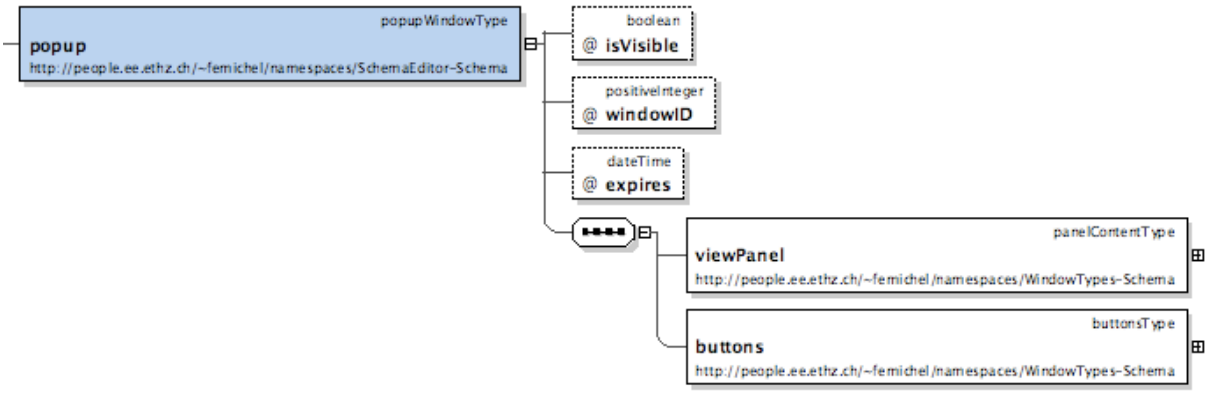
The question remains whether the visualization of a substitution group should focus on the place where elements actually are substitutable, rather than focusing on the place where the head element is declared. Put differently: In our example Schema, the content model of the global type *editorType* is basically the same from a conceptual point of view, regardless of whether the set of legal sub-elements is expressed by means of a choice model group enumerating all sub-elements or a substitution group containing these sub-elements. A visualization expressing this equality could be very useful, as it potentially hides XML Schema-specific particularities, providing a clearer *conceptual* view.

oXygen allows for references to be found easily. The search for references (and for definitions, vice versa) spans multiple schema documents, if the Schema is composed of multiple schema documents. This meets our expectation that such functionalities should focus on the Schema's data model rather than on the schema documents.

oXygen offers an alternative view of Schemas which is called *Logical Model View*. It provides a bird's view on the whole Schema, considering all Schema documents, and omitting much of XML Schema's syntax parts which are depicted in a very detailed way in the standard view. It basically only displays entities that can appear in instances, i.e. attributes and elements. While such a simplified view can be useful at revealing the structure of possible instance documents, it fails e.g. when the Schema is a library-like collection of types, such as the document *WindowTypes-Schema.xsd* of our guinea-pig Schema. Furthermore, the simplified view also omits parts that can be relevant for the model-perspective structure of a Schema. For instance, substitution groups are no longer indicated, and neither are substitutable elements, although such properties are essential for capturing the logical structure of a Schema.

One advantage of the simplified view is that the content models of derived types are resolved, thus showing the content model which is relevant for instance documents. The fol-

lowing figure contains a rendering of a model group that results from a type derivation by restriction.



The "Logical Model View" of oXygen, showing a more compact representation of an element whose type is derived by restriction. Note that components may have different namespaces; the view shows namespace URIs rather than on namespace prefixes.

Inherited attributes are – even though not present in the XML syntax – shown, while prohibited elements and attributes are not shown any longer. This indeed helps revealing the logical structures and avoids some of the problems caused by the XML syntax.

Building the simple Schema:

Because of the close proximity of the graphical notation to the XML syntax of Schema, graphically building schemas basically is writing a Schema document using right-click instead of writing pointy brackets.

The predominating input method are property boxes displayed upon context click. Unfortunately, clicking was not very reliable in our set-up, resulting in a somewhat annoying user experience. Selecting rows in such context boxes inserts the corresponding attributes into the Schema document. They remain in the source, even if no value has been selected. This results in invalid Schemas – either because the respective attribute is not allowed to be empty, or because specification of two attributes at one time is forbidden (e.g., *fixed* and *default*).

The editor does not enforce use of element names or references. It is possible to create *element* elements having neither a *name* nor a *ref* attribute set, which again invalidates the Schema.

The placement of some of the entries in those menus is counterintuitive, e.g. the (alphabetical) order of *minOccurs* and *maxOccurs*.

Undo operations in the graphical view are possible.

There is no obvious way for specifying namespaces. We had to use the *tree editor*, in fact a simple XML editor, and add the namespaces by inserting an attribute with name *xmlns:wt* and the appropriate value. In our opinion, this is disappointing, because tasks perhaps unfamiliar to a lesser experienced users like specifying namespaces are among those we would expect to be supported in a reasonable way by a graphical editing tool.

The tool apparently manages to resolve all imports when visualizing entities, but it does not list imported types in the selection menus used for defining base types and alike.

The tool allows one to insert both *extension* and *restrictions* within one complex type, which again results in a document that violates the XML Schema specification.

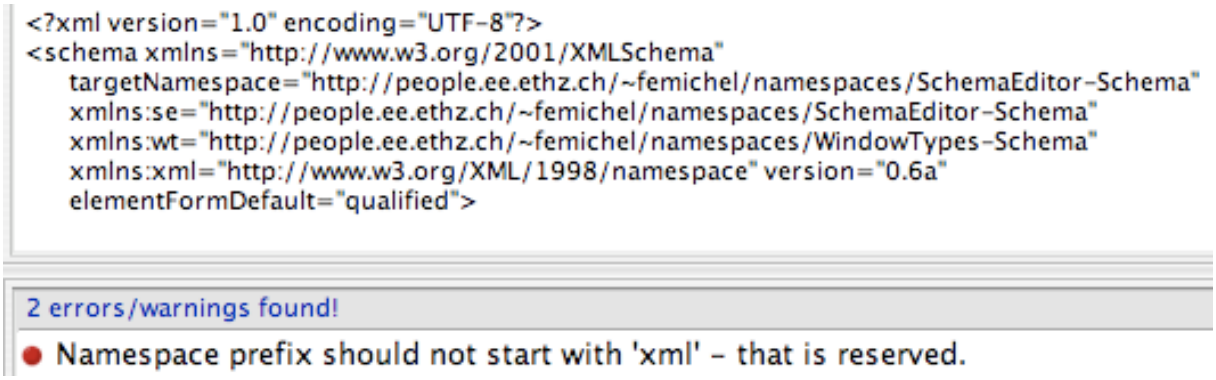
Documentation elements can be inserted, but not filled with text.

Identity constraints can be created, but again only in a manner very similar to source-editing. There is no support for building the XPaths.

5. *Turbo XML*

Tibco's Turbo XML is an element-centric editor, but with a visual formalism somewhat different from other common tools.

When opening our Guinea Pig Schema, the tool has problems with the namespace declaration *xmlns:xml="www.w3.org/XML/1998/namespace"*. It complains:

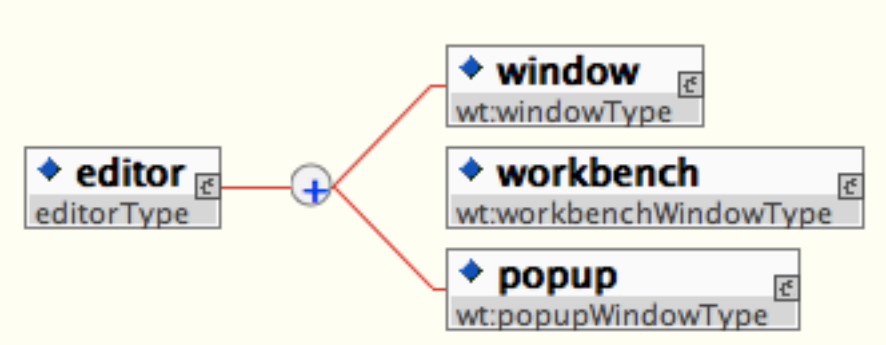


Turbo XML claiming 'xml' *to be a "reserved" namespace prefix.*

According to the specification [xmlns], the prefix "xml" is bound to the namespace "http://www.w3.org/XML/1998/namespace", but an explicit declaration is not illegal. The specification says: "It may, but need not, be declared". However, the problem is considered to be an error rather than a warning and requires the offending lines to be removed from the Schema files. Afterwards, parsing succeeds smoothly.

Whereas most of the other editors commonly represent the order constraints of the model groups (i.e., the differentiation choice / sequence / all group) as a box between parent and children elements (maybe with a little "+" / "-" box[17] indicating the ability to expand / collapse the model group's subtree), Turbo XML goes a slightly different way: There is no dedicated box for the model group. The order constraint rather is expressed by the way the branches of the subtree are drawn: Angular branches indicate sequences, straight lines stand for choices, and curved arcs represent all groups. Note that there are only arcs to the outmost child elements, not to each of them; the arcs form a *bracket* rather than a *tree*. This leads to some of the simplification of adjacent model groups we stated in section *Features to Test*.

[17] Akin to the node symbols in the tree views in Microsoft Window's Explorer.

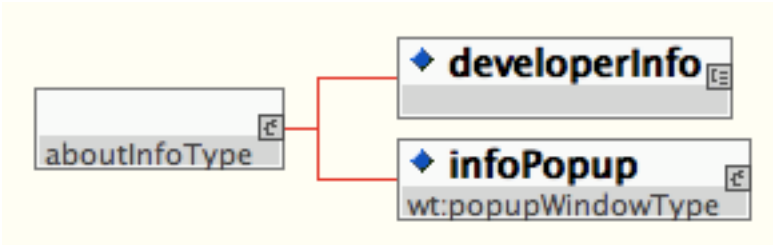


A choice model group in Turbo XML. The boxes are split vertically; the lower, grey-shaded part displays the element type, including a namespace prefix, if the type is imported . Note the little circle containing a "+" symbol: This indicates the model group's cardinality using Kleene operator syntax.

The functionality of the "+" / "-" boxes found in many other tools, i.e., the ability to unfold / collapse the subtree, here is provided by the little symbols within the parent element's box, on the right side, next to the red arc connecting the model group children elements. Note that these symbols not only are simple push-buttons for unfolding the subtree, but also give information whether the content model is of simple or complex content, and if the latter is the case, it indicates if the content is *mixed* or not.

Content model / simple types in Turbo XML. In the case of complex type / simple content, an appropriate simple type symbol is employed for the content model..

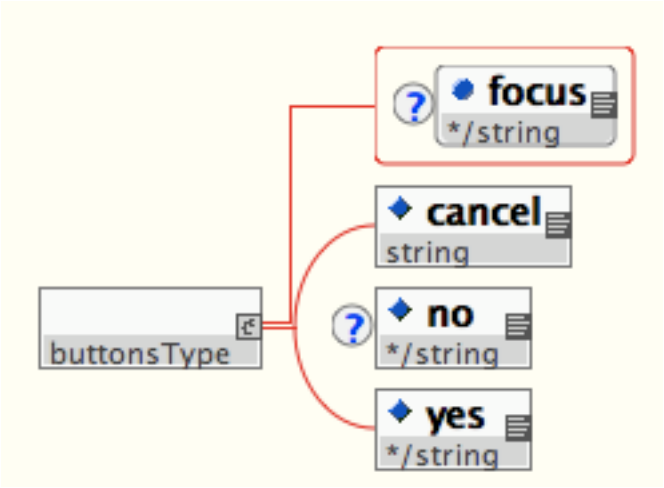
The model group also shows its cardinality by means of little circles at the splitting point of the red arcs: This circle either contains a *Kleene operator* giving the cardinality, or a range symbol ".." and the lower and upper bound written above and beneath the circle respectively[18].



A sequence *model group in Turbo XML. The left box represents a named type, the box on top right the contrary, an element using an anonymous type.*

Anonymous types are represented by leaving empty the lower, grey-shaded half of the elements boxes, usually filled with the element's type name. Named type declarations are handled the complementary way: The upper half normally used for the element's name is left blank, while the type's name is displayed in the lower half.

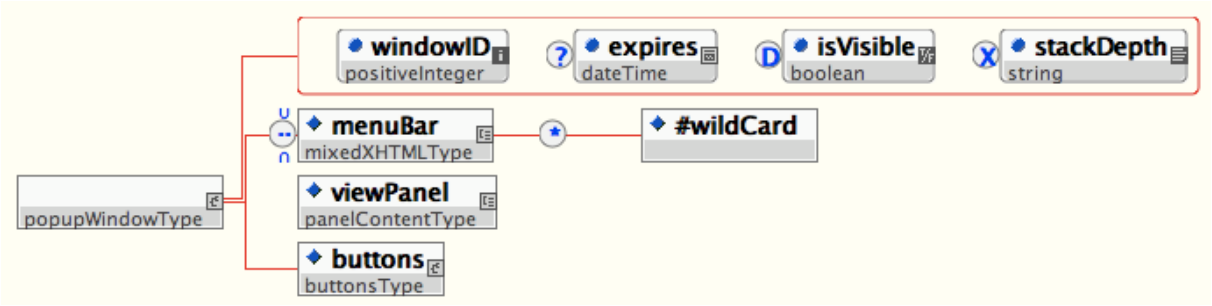
[18] Unfortunately, there seems to be a rendering problem here, clipping off parts of the numbers, resulting in somehow bizarre, symbol-resembling signs.



A all *model group in Turbo XML. Note the visualization of attributes as a specially outlined element child. Furthermore, one can see that the little circles containing Kleene operators can be prepended to single elements as well indicating their cardinality.*

Attributes are represented as children of the element they belong to. The discrimination from child elements is done by a red outline comprising all attributes and being attached to the parent element's box via a dedicated arc, and a blue disk rather than a blue diamond in the top left corner of their box.

The aforementioned circles containing Kleene operators can be prepended to any element or attribute, indicating their respective cardinality. In the latter case, obviously only the Kleene operator "?" is applicable, but the symbol vocabulary is extended to express special properties of attributes instead, e.g. if the attribute has a default value or prohibited use.



*Representation of multiple attributes with symbols indicating default values (*isVisible*), optional (*expires*) or prohibited (*stackDepth*) use. The topmost element (*menuBar*) shows a* [0..0]*-cardinality and wildcard content.*

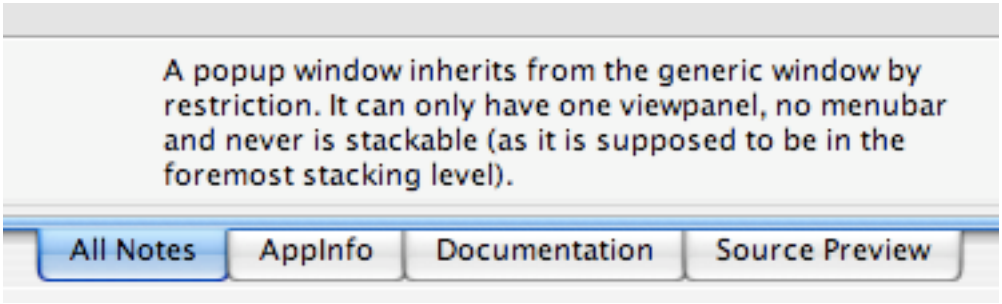
Type inheritance relationship is not indicated graphically. But a sidebar listing types in a spreadsheet-like manner shows base type and the kind of inheritance:

D	Complex Type	Restricts	Content	Content Model	Attributes
E	popupWindowType/menuBar		Type	mixedXHTMLType	
E	popupWindowType/viewPa...		Type	panelContentType	
E	popupWindowType/buttons		Type	buttonsType	
T	popupWindowType	r_windowType	Elements	(menuBar{0,0} , viewPanel , buttons)	isVisible, stackDepth

Spreadsheet-like sidebar listing types in Turbo XML.

Note that there is a column "Attributes" and a column "Content Model", where the content model is given in a DTD-like form using logical operators ("|", "&", ","), Kleene operators ("?", "+", "*") and range indicators similar to the syntax found in regular expressions ("{1..5}"). Here the "Attribute" column lists only the attributes repeated in the derived type's declaration (*popupWindowType* is a restriction of *windowType*). This is especially confusing as the attribute *stackDepth* is only repeated to be marked *prohibited*. Listing this attribute, but leaving out other ones not affected by the restriction draws a distorted picture.

Annotations are not shown in the graphical view itself. Instead, a dedicated pane lists all available annotations letting the user choose between *AppInfo* and *Documentation* content (a differentiation made by XML Schema) and a source preview of the respective element. This can be an interesting approach especially for elements with incomplete or no annotations[19].



The annotation pane in Turbo XML, including a source preview tab.

Substitution groups are neither resolved nor represented graphically. At least, a pane in a sidebar lists the property *substitutionGroup* of the elements affected.

Apparently, there is a way to generate documentation called "Schemadoc", but attempting to use the tool, it throws a JAVA error and quits.

Building the simple Schema:

The tool provides convenient support for importing namespaces: It offers a dialog box where one can select the desired schema document; according prefix and namespace are retreived from that and inserted automatically.

The attribute "elementDefault" is set to "qualified" by default. This is a very sensible choice, circumventing an arguable issue of the Schema specification. It again is one of such conveniences we expect an editor to provide.

As described above, the content models have to be specified in a DTD-ish syntax. Even though this leads to rapid development in case of users familiar with the DTDs syntax, it is not a fully graphical input mechanism we are looking for.

Complex types are not defined in the type pane, but in the element pane, which can be very confusing. Maybe there are conceptual reasons to treat complex types more like elements than simple types, but these reasons would have to be reflected in a more elaborate

[19] Or for developers adhering to the principle: *The source code is the best documentation.*

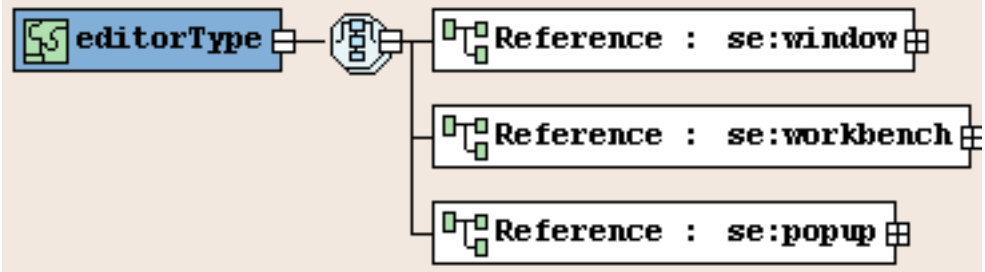
way in order to be as convincing as the evidence, that both are named "types", which strongly suggests to look for these features in the same place when using a tool.

Again, constraints cannot be specified.

The editor rewrites the documents edited, doing some kind of canonicalization, potentially heavily rewriting the document.

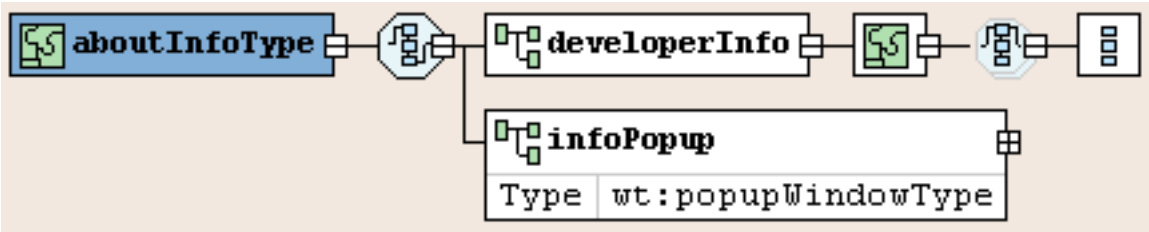
6. Stylus Studio

DataDirect's Stylus Studio is an element-centric editor. It resembles SyncRO's oXygen in the way it maps Schema's XML syntax onto a graphical representation. The model groups are visualized in the following way, using only slightly different symbols for *choice*, *sequence* and *all* model group:



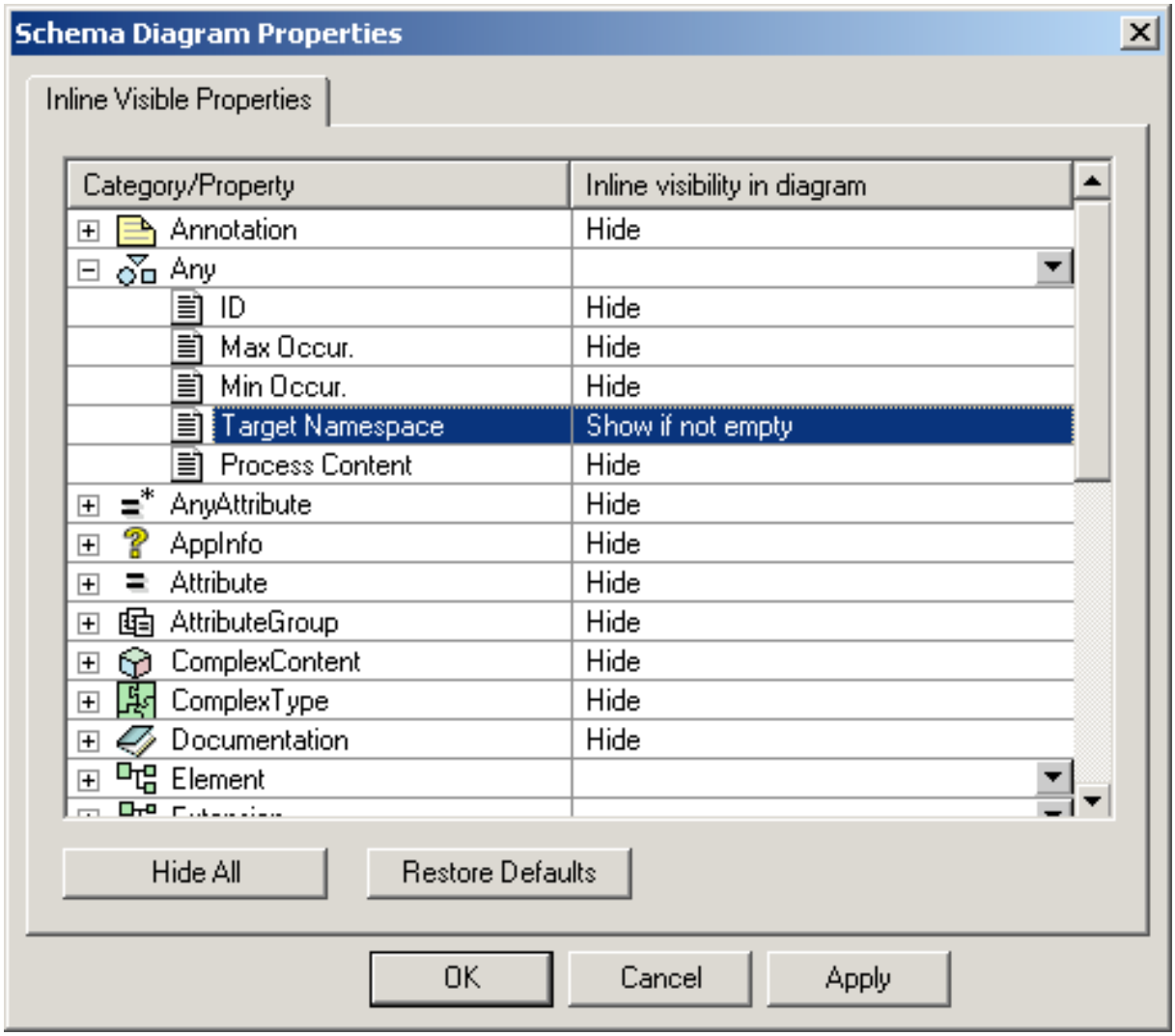
The choice *model group in Stylus Studio. The model group has set* maxOccurrence *to* "unbounded"*: This is expressed by a model group symbol shown in a stacked manner.*

The boxes only show element names by default, but additional information like type or base type can easily be switched on:



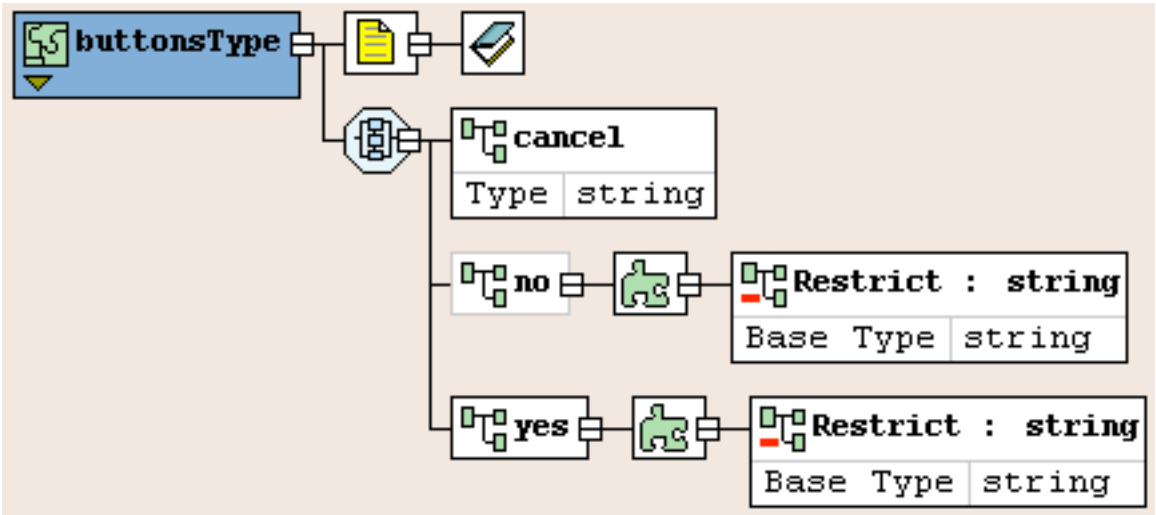
The sequence *model group in Stylus Studio. The element* infoPopup *shows the visualization of an element using a global type, for the element* developerInfo *it can be seen how anonymous complex types, mixed content and wildcard elements are represented.*

Although the wildcard element in the example above only allows elements from a specific namespace, this namespace is not shown. However, the visibility of this information can be edited (besides the display properties of many other attributes) in a special dialog. The target namespace of the schema for example can be set to "Show if not empty":



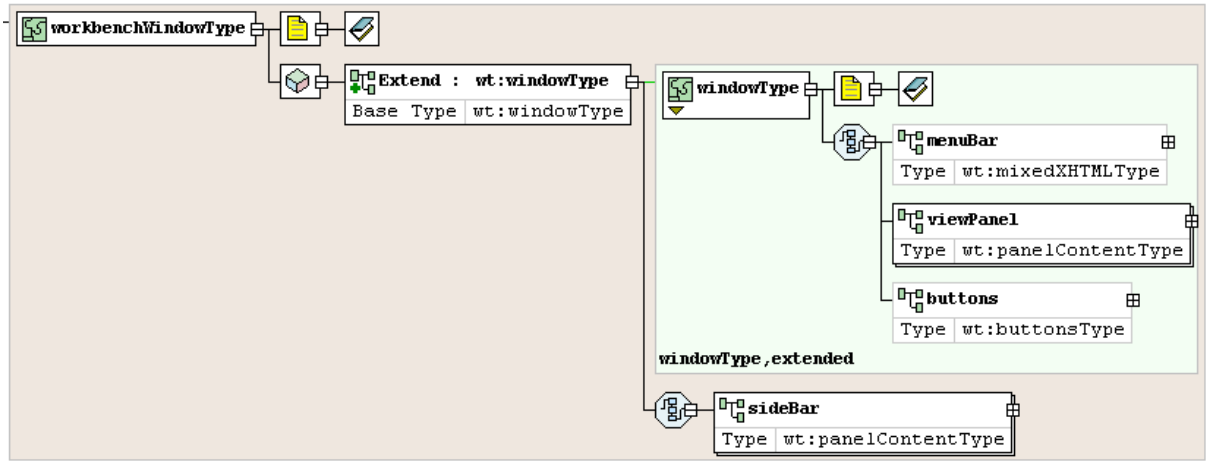
Stylus Studio's property dialog, where display of a vast number of Schema attributes can be configured.

Finally, the *all* model group looks as follows:

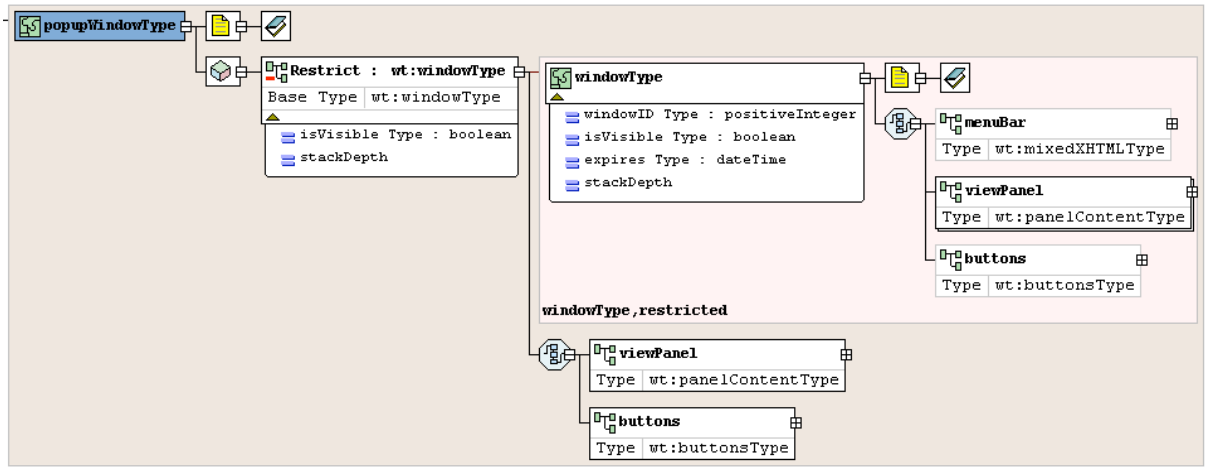


The all *model group in Stylus Studio. Note how optional elements like the element "no" are shown in grey boxes.*

Type derivation is rendered in a way very similar to oXygen, with the same criticism applicable to Stylus Studio as well. *Extension* and restriction are visualized in the following ways:

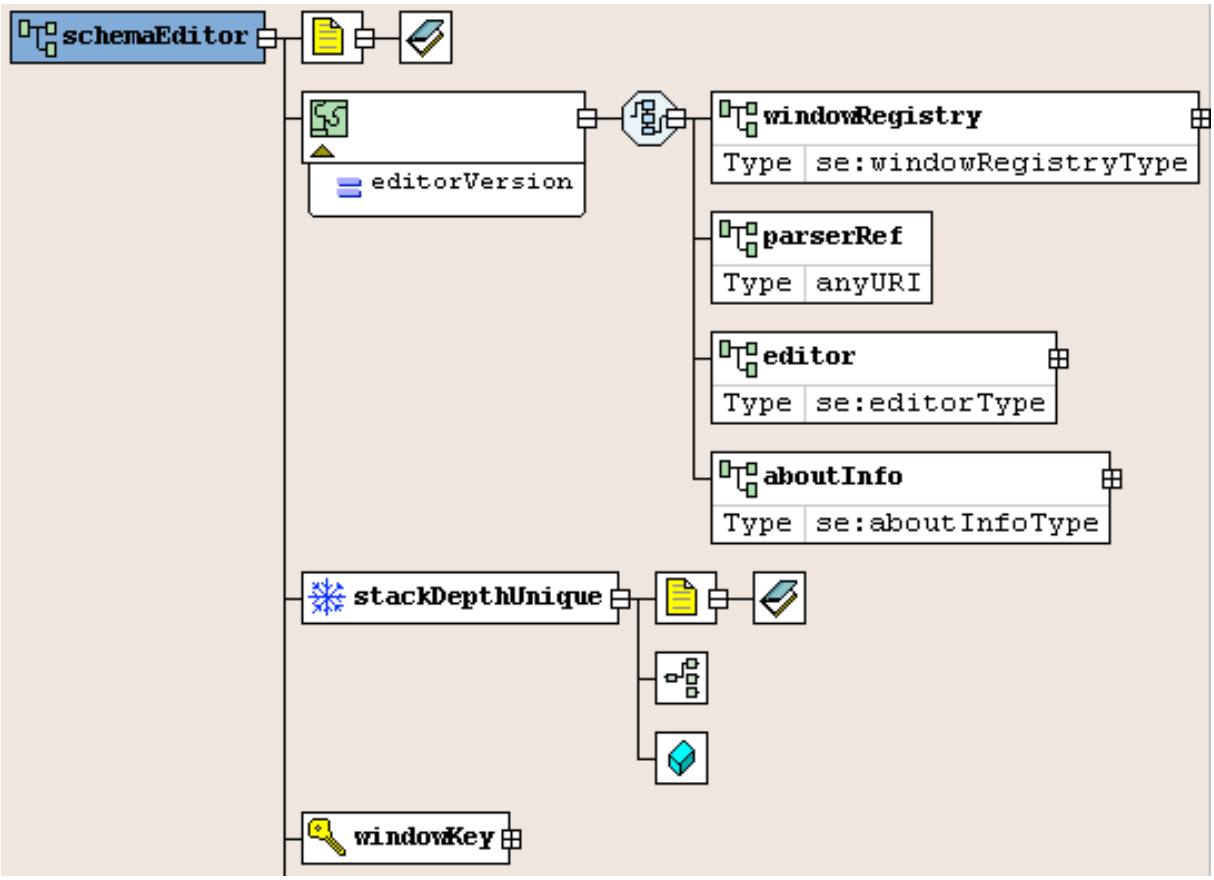


Type extension *in Stylus Studio. The model group appended by the extension is shown connected to a box representing the extension type.*



Type restriction *in Stylus Studio. Again this is a mere mapping of the Schema document.*

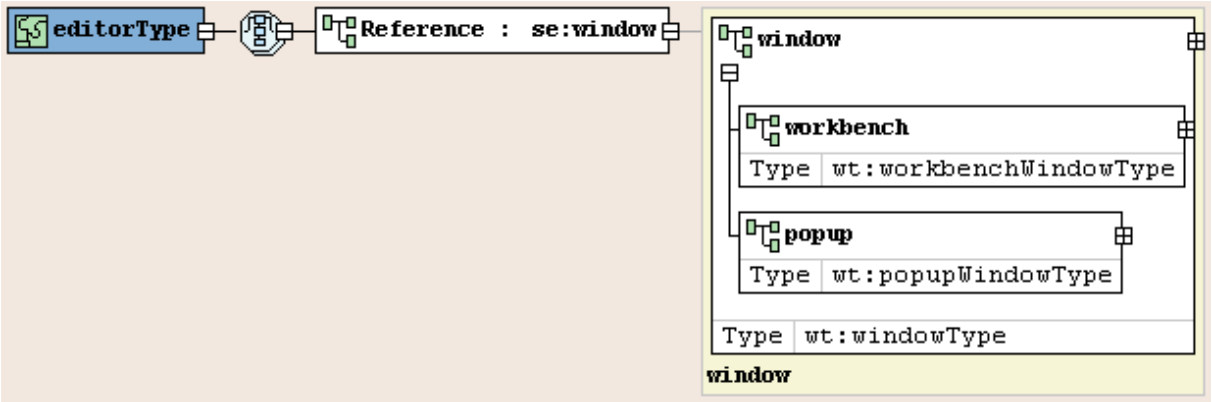
The symbols representing complex types are very similar to the ones used for simple types; in fact, the latter is represented by a piece of a jigsaw puzzle, while the former shows a symbol of multiple such pieces, connected to each other. This might be a nice idea reflecting the "complex" properties in its literal sense. For practical use however, a more clear graphical distinction might be beneficial.



Display of elements defining identity constraints *in Stylus Studio. Note that attributes (like* editorVersion*) are attached to the box representing the anonymous complex type rather than to the element's box.*

Having a visual symbolism close to the Schema's XML syntax, constraints are graphically shown, but only in terms of the elements defining them.

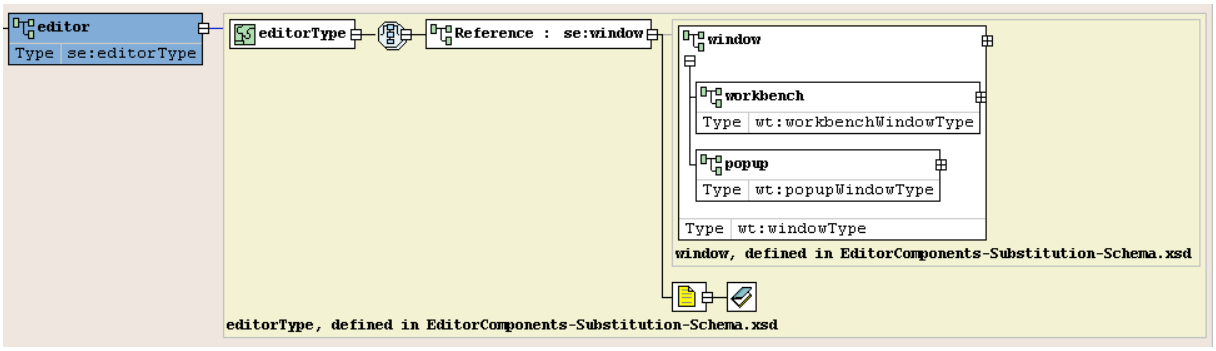
Substitution groups are recognized and resolved; a little, free-floating expansion box within the group's head element reveals the group members:



Type substitution in Stylus Studio. A little "+"-box within the substitutable element reveals the other members of the substitution group.

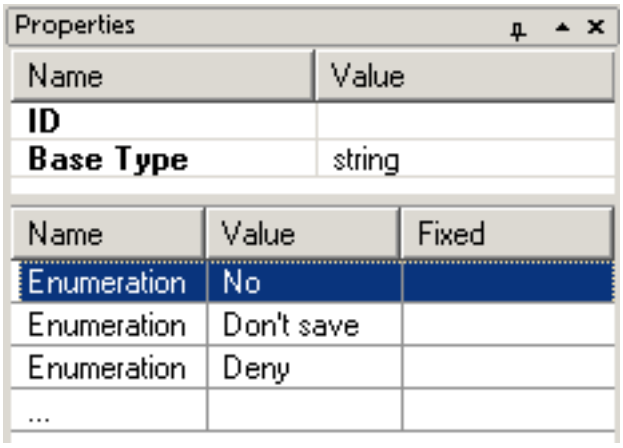
When used in a bigger context, this approach leads to a very lengthy visualization of a fact structurally very similar to a simple choice model group (we discussed this before). If we look at the following graphic while keeping in mind the sleek representation in XML Spy, we

clearly can see how weakly the structural or functional meaning of the data model is expressed by the visualization:



A head element of a substitution group as depicted in Stylus Studio. The designer's intention may have been to use the substitution group as a functional equivalent of a choice model group. This fact is more obscured rather than emphasized by the visualization strategy.

Additional properties of attributes are easily visible in an overview sidebar:



Property sidebar showing the value list of a simple type restricted by enumeration.

Building the simple Schema:

If one defines (for there is no default one) a target namespace (after having enabled view of the corresponding attribute or using the *Properties* sidebar), Stylus Studio at least inserts a corresponding default XML namespace.

There is some support for *importing* Schemas, but the wizard creates absolute path names for the *schemaLocation* and does not provide any possibility to specify a namespace prefix. If a prefix more descriptive than *auto1* is desired, one has to change that in the source.

Attribute types and default values cannot be defined by purely graphical editing, but the property sidebar mentioned before provides a convenient way for doing so.

Because Stylus Studio again is following a visualization approach that simply maps the elements of a Schema document to boxes, it is possible to specify all identity constraints required by the *pseudoSchema* above. But there is no support at all when inserting the elements defining the constraints. The designer has to know all the – partly intricate – particularities (namely: where to declare those elements, the inner structure of those elements, connecting *keyRefs* to *keys*, choosing the right selector set, building correct XPath

expressions). In fact, it not only is possible to define *key* elements with the wrong order of *selector* and *field* sub-elements, *keys* defined in such a manner even pass the built-in validation.

The context menu appearing when right-clicking elements includes a way to copy XPath expressions identifying them. One could think that this perhaps eases the task of assembling correct XPath expressions. But on the one hand, path expressions selecting more complex node sets still have to be built or concatenated by hand, and on the other hand, the path expressions built are very verbose (using *local-name* and *namespace-uri* instead of prefixes[20]) and therefore hard to understand for the less experienced user seeking for support.

5.2. The Matrix

The following matrix gives a more strict, tabular overview. The rows of the table correspond to the *Features to Test* defined above.

[20] This is not a bad idea *per se*; in fact, not relying on prefixes and the values of **FormDefault* attributes makes the XPath expressions more robust to changes. But here, we are investigating it in the context of a GUI supposed to ease XPath construction.

	WST 1.0.3	WST 1.5.0	XML Spy
Model groups	No simplifications	No simplifications	No simplification
Mixed content	Not indicated	Not indicated	Indicated
Types	List in sidebar, boxes around content model, containing type name	Type-centric visualization; List in sidebar	List in sidebar, boxes around types, global types as root of partial trees selectable
Type inheritance	Extension: Inherited content model in a dashed box. No graphical connection to base type.	Shown, but arrows not differentiating type of inheritance	Boxes around inherited content model, grey-shaded restricted parts, cardinalities of the base type's content model etc. Even more details when selecting the derived type directly (control elements, attribute restrictions, base type name)
Substitution	Not shown	Not shown	Shown; rendered similarly to choice group
Constraints	Not shown	Not shown	Shows constraint defining elements, but not constraints
Annotations	Sidebar, text shown on focus	Sidebar, text shown on focus	Annotation contents embedded within the graphical view
Invisible properties	Weak support. Inherited attributes not shown	Weak support (some restrictions in sidebar)	Dedicated facet panes etc. GUI display highly configurable via "Schema settings"
GUI editing	Limited. No constraints, complex types with simple content.	Limited & erroneous. No constraints.	Powerful, but some things still need the Grid View. Weak support for imports and constraints
Documentation	No	No	Rich, configurable hypertext documentation including source fragments and graphics
Foundations	Ad hoc	Ad hoc	Ad hoc

Comparison table listing the various Features to test *specified in section* Comparison Set-Up

	oXygen	Turbo XML	Stylus Studio
Model groups	No simplifications	Symbolism leads to some simplification	No simplifications
Mixed content	Not indicated	Indicated	Not indicated
Types	Separate pane listing the current documents's types, boxes similar to WST 1.0.3	Tabular type perspective. Apparently not working.	No type-oriented views or trees.
Type inheritance	Base types in separate boxes, that are connected to the line between the boxes depicting inheritance role and model group	Not shown. A spreadsheet-like tabular view lists base type and kind of inheritance for derived types	Dedicated boxes representing inheritance, listing the base type's name. Base type (in a green box) and additional or repeated parts of the content model are attached to this box.
Substitution	Shown at head declaration	Not shown	Group members are listed under a free-floating "+"-box within the group head's box
Constraints	Shows constraint-defining elements, but not the constraints	Not shown	Shows constraint-defining elements, but not the constraints
Annotations	Elements shown in tree, contents in source pane	Dedicated pane differentiating between appInfo, documentation and source view	Elements shown, contents unfolded depending on a button in a menu bar.
Invisible properties	Almost everything shown in tree; multiple sidebars available	Different, redundant sidebars & perspectives.	Neat overview in the sidepane; GUI display highly configurable
GUI editing	Powerful, but very close to source editing. Weak namespace support.	No constraints. Content model has to be typed DTD-like. Handy support for imports.	Very close to source editing. Allows for erroneous Schemas to be created.
Documentation	Rich, configurable hypertext documentation including graphics, source as well as a legend and glossary	Available, but not working	In-window documentation view including source, graphics, and glossary.
Foundations	Ad hoc	Ad hoc (DTDs)	Ad hoc

Comparison table listing the various Features to test *specified in section* Comparison Set-Up

5.3. Special observations

5.3.1. Common Problems

Almost all editors under test failed at revealing the actual structure (i.e.,the resulting content model and set of allowed attributes) of types derived by restriction, with XML Spy being the only exception.

None of the editors investigated provided substantial support for building or visualizing identity constraints: Some of them do not display identity constraints at all (1,2,5), the other merely display the definition elements of the identity constraints instead of their effect (3,4,6).

Although some editors offer different *views*, these views are usually not the different perspectives on the data model we described as point 2.3 of our list of *Criteria*. Some of the *views* available are simply XML editors (perhaps slightly adapted for use with XML Schema documents), and as a consequence, the structure they display is the tree of the Infoset of the XML representation. This clearly is not a visualization of the XML Schema structure that we are looking for. Other *views* only let the user choose another level of granularity. While this can prove beneficial for keeping large Schemas displayable, it does not help revealing the structures of the Schema or focusing on different concepts of the data model. Most notably, such kinds of *views* offer no support for coping with both element and type perspective.

Substitution groups are not shown by several editors (1,2,5), and the visualization strategies of the other editors (3,4,6) are of varying usefulness depending on the context of use and the level of proficiency of the user of the tool. The next section discusses this in more detail.

Overall it can be said that exactly the problematic parts of XML Schema (that we hoped to be addressed by sophisticated visualizations) also are the parts where the editors show the most problems and deficiencies. The lacking, erroneous, or insufficient support for XML namespaces completes this discovery.

5.3.2. Promising Approaches

The visualization of *type restriction* in XML Spy may serve as an example of how visualization can alleviate use and understanding of advanced features of XML Schema, thereby hiding difficulties introduced by the XML syntax of XML Schema. We imagine that equally strong support for other problematic ares of XML Schema like identity constraints and substitution groups would be very beneficial and desirable.

Nevertheless the graphical formalism of XML Spy's representation still is not that easily intelligible. This is due to the level of detail and — because it allows for editing the Schema by means of the closing boxes etc. explained above — control. Maybe the trade-off between level of detail and visual simplicity could be reconciled by introduction of different views.

A different view (at least different from other editors) offers WST 1.5.0 with its decision of using the type hierarchy as the principal structure of its visualization. This is a promising approach for different reasons: First, the type hierarchy is well-suited for visualization, because it forms a tree. This does not hold for the element structure that cannot be represented by one single tree, as it expresses a grammar that describes a *class of trees*. Sec-

ond, type derivation is one of the advanced features of XML Schema that we identified as one of the parts where support through a good visualization would be helpful. And finally, visualizing the type hierarchy can help to recover an essential part of the underlying model.

However, the visualization of the type hierarchy in WST 1.5.0 unfortunately is very limited. The restriction to only two tiers at a time does not allow for displaying the whole type hierarchy. Even worse, it makes it almost impossible to navigate the type hierarchy. And only the combination with an element-oriented perspective would allow to fully exploit the concept of views.

The *Logical Model View* of oXygen partially satisfies our requirement of a model-oriented view, but it also hides properties that are essential for the logical structure (such as substitution groups), and it fails at visualizing types and their relationships. However, it allows for whole Schemas to be viewed (even if components are imported from different namespaces), and it may prove very helpful when creating instances conforming to this Schema, because it focusses on visualizing the structure of possible instance documents.

Different views are not only helpful in the case of types. Different scenarios of use can be supported by specialized views as well. Substitution groups may serve as an example here: It is hard to decide whether the approach chosen by XML Spy or the one of oXygen is better *per se*. It depends on the context of use. For developing Schemas, the latter approach can be fine: A developer familiar with XML Schema needs to see which elements are using a given element as their substitution group head. But from perspective of using the Schema, one probably is more interested in a model group-like view, where the question is not "to which substitution group does this element belong". The only interesting fact is "which elements can occur at this point in the model group (i.e., substituting for a given element)". The latter situation very much resembles the concept of a choice model group, therefore the approach taken in XML Spy might be preferable.

Speaking of content models, Turbo XML's non-graphical input method has to be mentioned. Even though not a visualization, the short and DTD-like notation that uses parentheses and Kleene operators can satisfy many of the requirements we demanded a visualization to fulfill: Good overview, simplified input, and hiding of particularities that are due to the XML syntax of XML Schema. From this we can conclude that the problem of how to support development of, reasoning about, and documentation for XML Schema is not limited to visualization. Other innovative ideas can contribute their parts to a solution as well.

6. Conclusions

The visualizations found in today's XML Schema editors often focus on the Schema *document* (the Schema's representation, more precisely the W3C's normative XML syntax), rather than on the Schema's model. Most of the editors do not address the data model at all and provide no interface to the model that is underlying the schema documents. It thus can be concluded that today's Schema editors view on XML Schema is still far from conceptual.

In general, we were disappointed by the weak features of the editors, particularly because many of the tools apparently have problems or very poor support for exactly those parts of XML Schema where we identified the biggest need for visualization support.

7. Bibliography

[wil06d] Arijit Sengupta and Erik Wilde, "The Case for Conceptual Modeling for XML", TIK Report 244, ETHZ, 2006

[moh05] Sriram Mohan and Arijit Sengupta, "Conceptual Modeling for XML — A Myth or Reality?" in Zongmin Ma (ed.), "Database Modeling for Industrial Data Management: Emerging Technologies and Applications", Idea Group Inc, 2005

[wil04i] Erik Wilde, "Metaschema Layering for XML", Workshop on XML Technologies for the Semantic Web (XSW 2004), Berlin, Germany, October 2004

[nec05] Martin Nečaský, "Conceptual Modeling for XML: A Survey", Praha, 2005

[xsd1] Henry S. Thompson, David Beech, Murray Maloney and Noah Mendelsohn: "XML Schema Part 1: Structures", Second Edition, W3C Recommendation 28 October 2004

[rng] James Clark and MURATA Makato, "RELAX NG Specification", OASIS Comittee Specification, December 3, 2001

[bex] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. "Expressiveness of XSDs: From Practice to Theory, There and Back Again." In Proceedings of the 14th International World Wide Web Conference, pages 712–721, Chiba, Japan, May 2005. ACM Press.formal-20010925, September 2001.

[xso] "XML Schema Object Model", https://xsom.dev.java.net/

[xmlns] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin: "Namespaces in XML 1.0", Second Edition, W3C Recommendation 16 August 2006, http://www.w3.org/TR/REC-xml-names/

[.net] "Microsoft .NET Homepage", http://www.microsoft.com/net/default.mspx

[j2ee] http://java.sun.com/javaee/

[xis] John Cowan and Richard Tobin: "XML Information Set", Second edition, W3C Recommendation 4 February 2004

[som] "XML Schema Object Model (SOM)", .Net Framework's Developer Guide, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconXSDSchemaObjectModelSOM.asp

[wst0] "XML Schema Infoset Model (XSD)", http://www.eclipse.org/xsd/

[sqc] "Schema Quality Checker" http://www.alphaworks.ibm.com/tech/xmlsqc

[cov1] "Schema/DTD Editors" on www.xml.com, http://www.xml.com/pub/pt/2

[xsd4] Section "Tools" on the W3C XML Schema web page, http://www.w3.org/XML/Schema#Tools

8. XML Schema Documents

8.1. WindowTypes-Schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://people.../WindowTypes-Schema"
      xmlns:wt="http://people.../WindowTypes-Schema"
      xmlns:xml="http://www.w3.org/XML/1998/namespace" version="0.5a"
      elementFormDefault="qualified">

      <annotation>
            <documentation xml:lang="en-US">
                  Some generic types representing different kinds of
                  application windows.
            </documentation>
      </annotation>

      <complexType name="mixedXHTMLType" mixed="true">
            <choice minOccurs="0" maxOccurs="unbounded">
                  <any namespace="http://www.w3.org/1999/xhtml"></any>
            </choice>
      </complexType>

      <complexType name="panelContentType" mixed="true">
            <choice minOccurs="0" maxOccurs="unbounded">
                  <any
                        namespace="http://www.w3.org/1999/xhtml
                                   http://www.w3.org/2000/svg">
                  </any>
            </choice>
            <attribute name="panelID" type="positiveInteger"
                  use="optional">
            </attribute>
      </complexType>

      <complexType name="buttonsType">
            <annotation>
                  <documentation xml:lang="en-US">
                        Buttons for user interaction. If there are buttons in
                        the first place, only the "no" button is optional.
                        "cancel" and a form of confirmation button must always
                        be present (unless it's supposed to be an aporetic
                        button selection).
                  </documentation>
            </annotation>
            <all>
                  <element name="cancel" type="string" fixed="Cancel">
                  </element>
                  <element name="no" minOccurs="0">
                        <simpleType>
                              <restriction base="string">
                                    <enumeration value="No"></enumeration>
                                    <enumeration value="Don't save"></enumeration>
                                    <enumeration value="Deny"></enumeration>
                              </restriction>
                        </simpleType>
                  </element>
                  <element name="yes">
                        <simpleType>
                              <restriction base="string">
                                    <enumeration value="Ok"></enumeration>
                                    <enumeration value="Yes"></enumeration>
                                    <enumeration value="Save"></enumeration>
                              </restriction>
```

```
                    </simpleType>
               </element>
          </all>
          <attribute name="focus">
               <simpleType>
                    <restriction base="string">
                         <enumeration value="cancel"></enumeration>
                         <enumeration value="no"></enumeration>
                         <enumeration value="ok"></enumeration>
                    </restriction>
               </simpleType>
          </attribute>
     </complexType>

     <complexType name="windowType">
          <annotation>
               <documentation xml:lang="en-US">
                    The basic type representing windows. defined as an
                    abstract type. (No, it isn't anymore. I changed my
                    mind.)
               </documentation>
          </annotation>

          <sequence>
               <element name="menuBar" minOccurs="0" maxOccurs="1"
                    type="wt:mixedXHTMLType">
               </element>

               <element name="viewPanel" minOccurs="1"
                    maxOccurs="unbounded" type="wt:panelContentType">
                    <annotation>
                         <documentation xml:lang="en-US">
                              A panel displaying graphical content encoded as
                              text/xml-svg.
                         </documentation>
                    </annotation>
               </element>

               <element name="buttons" minOccurs="0" maxOccurs="1"
                    type="wt:buttonsType">
               </element>

          </sequence>
          <attribute name="windowID" type="positiveInteger"
               use="required">
          </attribute>
          <attribute name="isVisible" type="boolean" default="false"></attribute>
          <attribute name="expires" type="dateTime"></attribute>
          <attribute name="stackDepth">
               <simpleType>
                    <restriction base="integer">
                         <maxInclusive value="0"></maxInclusive>
                    </restriction>
               </simpleType>
          </attribute>
     </complexType>

     <complexType name="popupWindowType">
          <annotation>
               <documentation xml:lang="en-US">
                    A popup window inherits from the generic window by
                    restriction. It can only have one viewpanel, no menubar
                    and never is stackable (as it is supposed to be in the
                    foremost stacking level).
               </documentation>
          </annotation>
          <complexContent>
               <restriction base="wt:windowType">
```

```
                    <sequence>
                        <element name="menuBar" minOccurs="0" maxOccurs="0"
                            type="wt:mixedXHTMLType">
                        </element>
                        <element name="viewPanel" minOccurs="1"
                            maxOccurs="1" type="wt:panelContentType">
                        </element>
                        <element name="buttons" minOccurs="1" maxOccurs="1"
                            type="wt:buttonsType">
                        </element>
                    </sequence>
                    <attribute name="isVisible" type="boolean"
                        default="false">
                    </attribute>
                    <attribute name="stackDepth" use="prohibited"></attribute>
                </restriction>
            </complexContent>
        </complexType>

        <complexType name="workbenchWindowType">
            <annotation>
                <documentation xml:lang="en-US">
                    A workbench window inherits from the generic window by
                    extension. It comprises a or some sidebar(s) which can
                    contain arbitrary content.
                </documentation>
            </annotation>
            <complexContent>
                <extension base="wt:windowType">
                    <sequence>
                        <element name="sideBar" minOccurs="1"
                            maxOccurs="unbounded" type="wt:panelContentType">
                        </element>
                    </sequence>
                </extension>
            </complexContent>
        </complexType>

</schema>
```

8.2. EditorCompmonent-Choice-Schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://people.../SchemaEditor-Schema"
      xmlns:se="http://people.../SchemaEditor-Schema"
      xmlns:wt="http://people.../WindowTypes-Schema"
      xmlns:xml="http://www.w3.org/XML/1998/namespace" version="0.6a"
      elementFormDefault="qualified">


      <annotation>
            <documentation xml:lang="en-US">
                  Some central components used for building the Schema editor
                  model in SchemaEditor-Schema.xsd, using the "choice model group"
                  approach.
            </documentation>
      </annotation>


      <!-- Imports, includes -->

      <import
            namespace="http://people.../WindowTypes-Schema"
            schemaLocation="WindowTypes-Schema.xsd">
            <annotation>
                  <documentation xml:lang="en-US">
                        Import the generic window types defined in
                        EditorWindows-Schema.xsd.
                  </documentation>
            </annotation>
      </import>

      <!-- Global type definitions -->

      <complexType name="editorType">
            <choice maxOccurs='unbounded'>
                  <element ref="se:window"></element>
                  <element ref="se:workbench"></element>
                  <element ref="se:popup"></element>
            </choice>
      </complexType>


      <!-- Global element definitions -->

      <element name="window" type="wt:windowType" block="#all">
            <annotation>
                  <documentation xml:lang="en-US">
                        A standard window.
                  </documentation>
            </annotation>
            <key name="windowPanelKey">
                  <selector xpath="./wt:viewPanel"></selector>
                  <field xpath="@panelID"></field>
            </key>
      </element>


      <element name="workbench" type="wt:workbenchWindowType" block="#all">
            <annotation>
                  <documentation xml:lang="en-US">
```

```
                    A workbench type window.

                    Within a workbench type window, sidebarIDs and panelIDs
                    must be present and locally unique, therefore we insert
                    the following identity constraints.
               </documentation>
          </annotation>
          <key name="workbenchPanelKey">
               <selector xpath="./wt:viewPanel|./wt:sideBar"></selector>
               <field xpath="@panelID"></field>
          </key>
     </element>


     <element name="popup" type="wt:popupWindowType" block="#all">
          <annotation>
               <documentation xml:lang="en-US">
                    A popup type window.
               </documentation>
          </annotation>
     </element>

</schema>
```

8.3. EditorComponent-Substitution-Schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://people.../SchemaEditor-Schema"
      xmlns:se="http://people.../SchemaEditor-Schema"
      xmlns:wt="http://people.../WindowTypes-Schema"
      xmlns:xml="http://www.w3.org/XML/1998/namespace" version="0.6a"
      elementFormDefault="qualified">


      <annotation>
            <documentation xml:lang="en-US">
                  Some central components used for building the Schema editor
                  model in SchemaEditor-Schema.xsd, using the "substitution
                  group" approach.
            </documentation>
      </annotation>


      <!-- Imports, includes -->

      <import
            namespace="http://people.../WindowTypes-Schema"
            schemaLocation="WindowTypes-Schema.xsd">
            <annotation>
                  <documentation xml:lang="en-US">
                        Import the generic window types defined in
                        EditorWindows-Schema.xsd.
                  </documentation>
            </annotation>
      </import>

      <!-- Global type definitions -->



      <complexType name="editorType">
            <choice maxOccurs='unbounded'>
                  <element ref="se:window"></element>
            </choice>
      </complexType>


      <!-- Global element definitions -->

      <element name="window" type="wt:windowType">
            <annotation>
                  <documentation xml:lang="en-US">
                        A standard window. This is the head of the substitution
                        group "wt:window".
                  </documentation>
            </annotation>
            <key name="windowPanelKey">
                  <selector xpath="./wt:viewPanel"></selector>
                  <field xpath="@panelID"></field>
            </key>
      </element>


      <element name="workbench" type="wt:workbenchWindowType"
```

```
            substitutionGroup="se:window">
            <annotation>
                  <documentation xml:lang="en-US">
                        A workbench type window. Member of the substitution
                        group "wt:window".

                        Within a workbench type window, sidebarIDs and panelIDs
                        must be present and locally unique, therefore we insert
                        the following identity constraints.
                  </documentation>
            </annotation>
            <key name="workbenchPanelKey">
                  <selector xpath="./wt:viewPanel|./wt:sideBar"></selector>
                  <field xpath="@panelID"></field>
            </key>
      </element>


      <element name="popup" type="wt:popupWindowType"
            substitutionGroup="se:window">
            <annotation>
                  <documentation xml:lang="en-US">
                        A popup type window. Member of the substitution group
                        "wt:window".
                  </documentation>
            </annotation>
      </element>

</schema>
```

8.4. SchemaEditor-Schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://people.../SchemaEditor-Schema"
      xmlns:se="http://people.../SchemaEditor-Schema"
      xmlns:wt="http://people.../WindowTypes-Schema"
      xmlns:xml="http://www.w3.org/XML/1998/namespace" version="0.6a"
      elementFormDefault="qualified">


      <annotation>
            <documentation xml:lang="en-US">
                  A sample file used for a comparsion among different Schema
                  editors.
            </documentation>
      </annotation>



      <!-- Imports, includes -->

      <import
            namespace="http://people.../WindowTypes-Schema"
            schemaLocation="WindowTypes-Schema.xsd">
            <annotation>
                  <documentation xml:lang="en-US">
                        Import the generic window types defined in
                        EditorWindows-Schema.xsd.
                  </documentation>
            </annotation>
      </import>

      <annotation>
            <documentation xml:lang="en-US">
                  There's a hard coded include switch that decides whether a
                  "choice" or a "substitution group" appoach shall be chosen
                  in order to model the se:editorType's content model.
            </documentation>
      </annotation>

      <include schemaLocation="EditorComponents-Choice-Schema.xsd"></include>

<!--
      <include
            schemaLocation="EditorComponents-Substitution-Schema.xsd">
      </include>
 -->

      <!-- Global type definitions -->

      <complexType name="windowRegistryType">
            <sequence maxOccurs="unbounded">
                  <element name="windowEntry">
                        <complexType>
                              <simpleContent>
                                    <extension base="string">
                                          <attribute name="ref"
                                                type="positiveInteger">
                                          </attribute>
                                          <attribute name="status" default="asleep">
                                                <simpleType>
```

```
                                        <restriction base="string">
                                            <enumeration
                                                value="running">
                                            </enumeration>
                                            <enumeration
                                                value="asleep">
                                            </enumeration>
                                            <enumeration
                                                value="blocking">
                                            </enumeration>
                                            <enumeration
                                                value="not responding">
                                            </enumeration>
                                        </restriction>
                                    </simpleType>
                                </attribute>
                            </extension>
                        </simpleContent>
                    </complexType>
                </element>
            </sequence>
            <attribute name="current" type="positiveInteger"></attribute>
        </complexType>


        <complexType name="aboutInfoType">
            <sequence>
                <element name="developerInfo">
                    <complexType mixed="true">
                        <choice minOccurs="0" maxOccurs="unbounded">
                            <any namespace="http://www.w3.org/1999/xhtml"></any>
                        </choice>
                    </complexType>
                </element>
                <element name="infoPopup" type="wt:popupWindowType"
                    block="#all">
                    <annotation>
                        <documentation xml:lang="en-US">
                            The popup showing information about the Schema
                            editor.
                        </documentation>
                    </annotation>
                </element>
            </sequence>
        </complexType>

        <!-- The global root element -->

        <element name="schemaEditor">
            <annotation>
                <documentation xml:lang="en-US">
                    As XML Schema has no way to denote an element as
                    root, we do it in this very documentary annotation:

                    This shall be the root element.
                </documentation>
            </annotation>
            <complexType>
                <sequence>
                    <element name="windowRegistry"
                        type="se:windowRegistryType">
                    </element>
                    <element name="parserRef" type="anyURI"></element>
                    <element name="editor" type="se:editorType"></element>
                    <element name="aboutInfo" type="se:aboutInfoType"></element>
                </sequence>
                <attribute name="editorVersion" use="required">
                    <annotation>
                        <documentation xml:lang="fr-FR">
```

```
                    Peut-être ça paraît trop rigoreux, mais en
                    essaiant d'amener le developeur à ne définer que
                    des nombre de version très brefs et concis, nous
                    demandons le developeur d'obéir à cette forme
                    stricte.
                </documentation>
            </annotation>
            <simpleType>
                <restriction base="string">
                    <pattern value="v\d+\.\d"></pattern>
                </restriction>
            </simpleType>
        </attribute>
    </complexType>


    <!-- Identity cpnstraints -->

    <unique name="stackDepthUnique">
        <annotation>
            <documentation xml:lang="en-US">
                The windows shall be neatly stacked. Each stacking
                level can only be assigned to one window at a given
                time.

                Here we don't include popups into the selector path,
                as the attribute "stackDepth" is prohibited for
                popups anyway. (To be more precise: actually we
                would have to make sure that when a popup is open
                (always in the foreground = stacking level 0) no
                other window has a stacking level of zero assigned.
                But this would involve co-constraints...)
            </documentation>
        </annotation>
        <selector xpath="./se:editor/*"></selector>
        <field xpath="@stackDepth"></field>
    </unique>

    <key name="windowKey">
        <annotation>
            <documentation xml:lang="en-US">
                @windowID must be a globally unique ID.
            </documentation>
        </annotation>
        <selector
            xpath="./se:editor/*|./se:aboutInfo/se:infoPopup">
        </selector>
        <field xpath="@windowID"></field>
    </key>
    <key name="reverseWindowKey">
        <annotation>
            <documentation xml:lang="en-US">
                This the reverse of windowKey in order to make the
                relationship window --- registry entry a one-to-one
                bidirectional one.
            </documentation>
        </annotation>
        <selector xpath="./se:windowRegistry/se:windowEntry"></selector>
        <field xpath="@ref"></field>
    </key>

    <keyref name="windowRegistryRef" refer="se:windowKey">
        <selector xpath="./se:windowRegistry/se:windowEntry"></selector>
        <field xpath="@ref"></field>
    </keyref>
    <keyref name="currentWindowRef" refer="se:windowKey">
        <selector xpath="./se:windowRegistry"></selector>
        <field xpath="@current"></field>
```

```
            </keyref>
            <keyref name="reverseWindowKeyRef"
                  refer="se:reverseWindowKey">
                  <selector
                        xpath="./se:editor/*|./se:aboutInfo/se:infoPopup">
                  </selector>
                  <field xpath="@windowID"></field>
            </keyref>
      </element>

</schema>
```