Small-scale XMI programming: a revolution in UML tool use?

Perdita Stevens
Division of Informatics
University of Edinburgh

October 23, 2002

Abstract

The main motivation for the development of the OMG's XML Metadata Interchange Format (XMI) was the need to have a standard way for UML tools to interchange UML models. However, the fact that UML models, saved as XML documents, are now accessible to standard XML tools, opens up new possibilities to which less attention has so far been paid. This paper discusses how XMI can be exploited for performing model analysis and housekeeping tasks and for the integration of third party or in-house tools. With the help of examples we focus on what can be achieved with small-scale XMI programming by a single developer in hours rather than days. We suggest that this new capability may have important and positive implications for the future of UML modelling using tools.

Keywords: XMI, XML, UML, model analysis, tool integration

1 Introduction and scope

UML, the Unified Modeling Language [6], is a standard diagrammatic language for recording the design of systems, especially object-oriented software systems. One of the main benefits of a *unified* modelling language is that it enables competition between tool vendors and allows users a wide choice of tools. Getting the most out of a tool – which is often a significant investment

– means using it as more than a fancy drawing tool. In this paper I will argue that the combination of XML and UML – especially in the form of XMI, the OMG's standard XML Metadata Interchange format [5, 9] – is crucial enabling technology for getting good value out of a UML tool. Indeed, I believe that it may *revolutionise the use of modelling tools in future.*

This paper focuses, as the title implies, on the use of XMI within a particular class of contexts. We are concerned with small-scale programming using XMI, and the mini-tools we will consider are things which can sensibly be built by a single person in a few hours (provided of course that that person has the appropriate skills, which will be discussed below in Section 5.2). The reason for this choice of focus is that, whilst it is on the face of it obvious that such uses of XMI are possible, the implications of the possibility do not seem to have received any attention in the UML community.

In contrast, the uses of XMI in more elaborate tools and tool-sets are widely appreciated: XMI can be used to assist major commercial efforts in tool integration or tool development (see e.g.[2]), and it can enable significant efforts to take place without privileged access to the internals of commercial tools (see e.g.[12, 11], or the new project *Design Environment for Global Applications*, `http://www.omnys.it/degas/`). These medium to large scale ventures make an important, if incremental, difference to the overall power of tool sets available. Yet it is arguable that the most important impact of XMI may turn out to be in myriads of small-scale uses, because here developers acquire qualitatively new capabilities. That is, we suggest that it is in the context of small-scale uses of XMI that the resulting change of practice will eventually justify the term "revolution".

The people building these mini-tools are of course envisaged to be people to whom programming is natural: we do not claim that analysts with business backgrounds are about to start writing code. A large proportion of current UML users are developers with programming skills, and it seems clear that the proportion will increase, because UML is now a standard part of university education for computer scientists and software engineers. Modern methodologies (the Unified Process [10] and eXtreme Programming [1], for example) recognise explicitly that design is not completed before programming begins. Therefore, if modelling is to be a first-class part of the development process, it is necessary that programmers take an active part in modelling, and that their models evolve as the design they embody is changed. This provides the requirement "pull" which complements the technology "push" in encouraging the style of XMI programming discussed in

this paper.

The paper is structured as follows. In Section 2 we will introduce and motivate the idea of small-scale tool development and integration, and discuss the XMI technology. Sections 3 and 4 give examples. In Section 5 we consider the wider implications of using this technology; we discuss the impact on the development process, and we discuss how similar techniques may also enable larger efforts. Section 6 discusses some of the technical pitfalls of the current XMI standard, and Section 7 concludes.

2 Small-scale tool development and integration

2.1 Tractability of code compared with models

Let us compare the facilities long available to the programmer with those available to the designer until now. Coders have the culture that they are in control; their tools are there to assist. The artefact they are producing – the code – is visible and available to them as plain text, so all text-manipulation facilities can be applied to code. For example, the developer can load the code into a more or less intelligent editor and manipulate it, either with a specific plan or in an attempt to understand it; they can apply editing macros; they can write scripts in their favourite text manipulation language, e.g. Perl or Python, to do what they will with the code. Any professional software engineer is used to doing this kind of manipulation on their code, and sometimes on other people's, on an ad hoc basis. For example, some of the tasks I have carried out personally are:

- "grep"ing for particular relevant identifiers, or for other patterns, in order to find relevant sections of code quickly;
- writing small scripts to check for, and in some cases correct, errors not caught by the type checker of the language, or violations of coding standards;
- extracting comments and formatting them for use in an in-house, non-standard help system whose use was mandated;
- extracting information about method signatures etc. for insertion in (LaTeX, in the case in point) documentation.

Any of these things could be done by hand, but, given the appropriate tools, can be automated *so easily that the effort of automation is often justified even for a one-off instance of the task.* Thus they fall into the category of mini-tools that may be developed, in the normal course of their job, by any developer, not just by those designated as project toolsmiths, let alone by specialised tool companies.

By contrast, until the advent of XMI there has been no reasonable way for modellers to analyse and manipulate their models in comparable ways. The primary view of the models was as diagrams on a screen, and there was no common textual format for storage of such models.

Note that an interesting feature of ad hoc utilities written for immediate use by the developer who wants them – whether they manipulate code or models – is that it is possible for them to be much simpler than a generic tool written by a third party to do the same task could be. This is because they can take advantage of any simplifying assumptions that can be made about *this particular* problem, e.g. things the developer happens to know about the nature of the code to which the script is to be applied. We will take full advantage of this in our examples (which are in any case simple for presentational reasons). As always, it is important that everyone involved understands the difference between an ad hoc script that solves the problem at hand and a robust tool that will work or fail gracefully in a wide variety of circumstances.

2.2 The XMI technology

When UML first appeared, there was no standard format for interchange of UML models; most individual tools had their own textual format which you could reverse engineer if you really wanted to, but the results tended to be unpredictable, because the formats had not been designed for such use. XMI, the OMG's XML Metadata Interchange format, is a vendor-independent format for saving and loading UML models. The ability to save and load XMI files is now a standard feature, provided by almost all major UML tools, from powerful (and expensive) modelling tools such as Rational Rose and Together to the cheaper drawing-oriented tools such as MagicDraw and QuickUML. Generally the tools offer to save models in XMI as a foreign format; however, some tools, such as the open-source modelling tool Argo/UML and its sister commercial tool Poseidon, use XMI as their native format. (Some incompatibilities between XMI written by different

tools still exist – we revisit the issue in Section 6 – but this will settle as XMI and the tools mature. All XMI files used with the scripts described here are written by Argo/UML 0.8 `http://www.argouml.org`.)

XMI is not an easy format for a human to read, and even small models can translate into large XMI files. However, the big advantage of XMI being based on XML is that the whole range of generic XML tools is available. Developers writing scripts to work on code generally avoid the need to parse the code, but scripts working on XMI can easily take advantage of parse tree information, because XML parsers are available in every popular language. The ability to analyse and manipulate XMI files means:

- Analyses or changes that used to be tedious to do by hand using the GUI of a UML tool can be automated; and so the temptation to let the UML model get out of step with the code or with other documentation is decreased. For example, changes made to the model may propagate to the code using tools' own forward/reverse engineering combinations, so that a developer may choose to make a change to the model and propagate it to the code rather than just changing the code.

- Any developer can write a script to extract information from XMI files and turn it into the input format of a proprietary tool they may be familiar with.

In the following sections we will illustrate each of these. We emphasise that we are interested in this paper in lightweight tool development and integration of the kind that can be done in minutes or hours, rather than days, weeks or months, by any developer.

3 Model analysis example

Any technology should make simple things simple. To start at the simplest conceivable level, consider a case in which a developer wishes to change some identifier wherever it occurs in the model. This can be done by loading the XMI file into any reasonable editor and applying search-and-replace capabilities, with or without querying the user for confirmation; or alternatively, by a script in any text manipulation language, such as Perl. The reason why this is worth saying at all is to illustrate that the problem faced by a developer wanting to manipulate a particular model is simpler than that faced by a

tool vendor wanting to allow such editing capabilities within their tool. For example, the developer may be confident that a simple "replace `retrieve` by `get`" will do, whereas a vendor writing a capability that should work on any model would have to worry more about the other contexts in which "retrieve" might occur and whether the replaced "get" version will clash with anything already in existence. The addition of human intelligence from the developer makes the problem soluble with much less programmed intelligence.

To take a slightly less trivial example, demonstrating what can be done with an XML parser, consider the case in which a developer wants to find all public attributes (but not operations) of classes, perhaps prior to replacing them with non-public attributes plus get/set methods for each such attribute. If the model is large this can be a tedious task to do graphically with a UML tool, or even with an unfamiliar proprietary scripting language. Figure 1 shows a Perl script that accomplishes it. Notice that although we use Perl within this paper because it is widely used by the users on whom we focus, the argument that such capabilities are potentially very useful holds equally for other languages – the possibilities are briefly discussed in Section 5.3 below.

However daunting this may look to non-Perl-devotees, this is quite straightforward to someone who knows Perl; it is the kind of thing that could be written in minutes (especially if this is not the first time the developer has used the XML::Parser module) and might save effort even if it was used only once.

The developer needs, of course, to understand the interface provided by the XML parser used. We briefly explain how this interface is used in the script.[1] The `parsefile` function returns a pair, which we refer to as a "tag-content pair". Its first entry is the name of the top-level XML element in the document parsed (`$tag`) and its second entry (`$content`) is itself a pair. Such a "content pair" (`$content`) for a given XML element has as its first entry a hashref containing the attributes of the XML element, and as its second a list of tag-content pairs, one for each of the children of the element. Thus the `parsefile` function in effect returns a tree containing all the relevant information about the document. The script consists, in the obvious way, of two mutually recursive functions which descend the tree.

[1] The commenting within the script is light, partly because provided the reader knows Perl and the parser it is very straightforward, and partly in order to fit the script onto one page.

Figure 1: Identifying public attributes

```
#!/usr/local/bin/perl
use XML::Parser;
$VISIBILITY = 'Foundation\.Core\.ModelElement\.visibility';
$CLASS = 'Foundation\.Core\.Class$';
$NAME = 'Foundation\.Core\.ModelElement\.name$';
$ATTRIBUTE = 'Foundation\.Core\.Attribute';

my $file = shift;
die "Can't find file \"$file\"" unless -f $file;
my $parser = new XML::Parser(Style => Tree, ErrorContext => 2);
my $pairref = $parser->parsefile($file);
mypair(undef, undef, @$pairref);

# take hashrefs for current class and attribute, plus tag-content pair
sub mypair {
  my ($recclass, $recattr, $tag, $content) = @_;
  return $content unless $tag; # deal with non-element nodes, i.e. text
  if ($tag=~/$ATTRIBUTE/) { # we've found an attribute.
    my $attr = {}; # represent it by a new anonymous hashref.
    # recurse to get name, class & visibility of this attribute
    myarray($recclass, $attr, @$content);
    print "Attr $$attr{name} in class $$recclass{name} is public\n"
      if $$attr{visibility} =~ 'public';
  }
  elsif ($tag=~/$VISIBILITY/ && $recattr)
    { $$recattr{visibility} = ${$$content[0]}{'xmi.value'}; }
  elsif ($tag=~/$NAME/ && $recattr){$$recattr{name} = @$content[2];}
  elsif ($tag=~/$CLASS/) {myarray({}, $recattr, @$content);}
  elsif ($tag=~/$NAME/) {
    $$recclass{name} = @$content[2] unless $$recclass{name};
  }
  else { myarray($recclass, $recattr, @$content); }
}

# take hashrefs for current class and attr, plus an element content, viz
# a hashref for attributes, which we ignore, followed by (tag,content)*
sub myarray {
  my ($class, $attr, $attributes, @rest) = @_;
  while (@rest) {
    mypair ($class, $attr, shift @rest, shift @rest);
  }
}
```

The first function, `mypair`, handles a tag-content pair, checking whether the XML element it records is of interest within the script and, if so, handling it appropriately. In some cases (such as encountering a `$NAME` element) there is no need to recurse further down the tree; the script simply extracts the relevant information (in this case the actual string which is the name) from the element and returns. In other cases (such as `$ATTRIBUTE`) information is needed from further down the tree. In these cases, `mypair` calls `myarray` to examine each of the child XML elements in turn. `myarray` does this simply by invoking `mypair` on each child. The presence of an anonymous hashref (as opposed to the undefined value, `undef`) as the second argument to either function is used with a dual purpose. First, it acts as a signal that the current element is inside an `$ATTRIBUTE` element of interest, which affects which information must be gathered. Second, it acts as a repository of information about that `$ATTRIBUTE` element, which is used after the recursive call to `myarray` terminates. Similarly, the first argument to the two functions records information about `$CLASS` elements.

As prefigured above, this script takes advantage of hypothesised simplicities of the target model; for example it takes no account of packages. Note also that the only component used is a simple XML parser. If one were going to do a lot of this kind of thing it would probably be worth writing a specialist XMI package, but even without one the script has only 35 (non-blank, non-comment) lines. It would not, of course, be hard to extend the script in various ways, for example to modify the XMI rather than merely examining it, if this was required. We chose to use a parser which returned a tree structure for purposes of illustration, but there are alternatives such as using a DOM-based parser; this provides more convenient access to individual elements, but may be considered less convenient when the structure of the document is more relevant.

4 Tool integration example

In this section we aim to back up our claims further by giving an example of how a UML tool that exports XMI can simply be integrated with another tool. The aim of the example is not to do clever tool integration – reports of much more impressive integrations have been published – but to show that something that may be worthwhile in context can be done very easily. The script described here was written in part of one afternoon, despite the

Figure 2: CWB input file (foo.cwb) produced by script

```
agent S3 = c.S1;
agent S4 = 0;
agent S1 = a.S2 + b.S3;
agent S2 = b.S3 + d.S4;
```

programmer's patchy acquaintance with the XML parser module.

A script very similar to the one shown above (but a little longer: 136 lines; see `http://www.dcs.ed.ac.uk/home/pxs/XSE2001/state.pl`) is used to extract the structure of any single flat FSM-like state diagram (no concurrency, actions etc.; though again, extending the script to more features would not be hard) and translate it into a Calculus of Communicating Systems (CCS) agent described in the input format of the Edinburgh Concurrency Workbench (CWB, see `http://www.dcs.ed.ac.uk/home/cwb`). The CWB is a broad verification tool (over 60 commands) in which many kinds of analysis can be carried out on such CCS agents, including model checking the full modal mu calculus with game-based feedback and many versions of equivalence and preorder checking.

Figures 2, 3 and 4 show an example of a UML state diagram drawn in Argo/UML, the CWB input file produced by the script from the saved XMI file, and a few CWB commands carried out on that input file (comments in [square brackets]).

Another, more complex, tool integration example is described in [4]. In this case information from UML state diagrams, augmented with some performance information, was combined with information from a collaboration diagram. One script produced an input file for a performance prediction tool, the PEPA Workbench, which calculated performance information for the input UML model. Another script incorporated the calculated information back into the original UML model, where it was presented to the user as annotations on the model.

Figure 3: State diagram for class Foo, drawn in Argo/UML

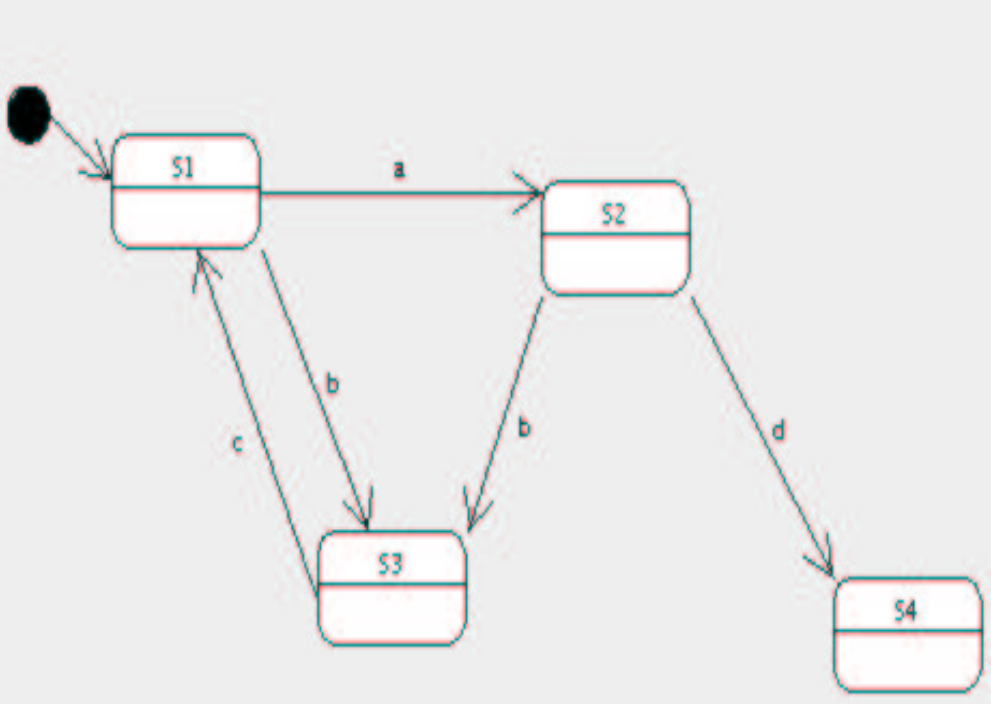


Figure 4: Using the CWB to analyse the behaviour

```
Edinburgh Concurrency Workbench, version 7.1,
Sun Jul 18 21:19:30 1999
Process algebra: CCS
Optional modules in this build: AgentExtra,Graph,Divergence,Contraction,
Equivalences,Logic,Simulation,Testing
Command: input "foo.cwb";
Command: deadlocks S1;
--- a d ---> S4
Command: deadlocks S3;
--- c a d ---> S4
Command: checkprop (S1, max (X.<->X));
true [i.e. there is an infinite path from S1]
Would you like to play (and lose!) a game against the CWB? (y or n) y
[output omitted]
Command: agent T = a.b.c.T + b.c.T;
Command: dftrace (S1,T);
Agent
S1
can perform action sequence
a,d
which agent
T
cannot.
Command:
```

5 Discussion and wider implications

5.1 The role of individual use of XMI in the development process

Should managers of software developers be enthusiastic, worried or both if individuals or teams which they manage want to make use of XMI technology in ways suggested here? Reasons for worry might include

- a fear that developing an XMI-based solution will take longer than the manual alternative and thus reduce productivity;
- a fear that developers will prefer "XMI hacking" to "real work";
- a fear that the mini-tools developed will contain bugs with possibly serious consequences;
- a fear that the organisation will be left dependent on a library of legacy Perl scripts or similar.

None of these fears is wholly groundless, but neither is any of them unique to this technology, and they seem to be manageable by standard processes, as follows. It is of course important that the developers concerned be conscientious and professional people interested in learning how to make appropriate use of new technology; but if this is not the case the organisation has greater problems than the use of XMI. Handling the first two fears requires a basis of professional trust between developers and their managers; it will take time for developers to establish an understanding of when an XMI-based mini-tool is appropriate and how quickly it can really be developed, and management may need to encourage caution at first. Fears like the last two arise in the context of any internal development of tools, and many organisations will already have processes to handle them. One approach may be to say that a developer may develop tools for his or her own exclusive use whenever, in his/her judgement, it is efficient to do so, but that any tool which is used by more than one person must go through a suitable quality assessment. This might involve, for example, ensuring that the tool has been adequately tested on data which is stored in the configuration management system along with the tool's source code, and that it is appropriately documented; especially, that any assumptions it makes about the models on which it works are made explicit.

For a comparison we may consider several other recently popular techniques which are sometimes resisted because of fears that they may reduce productivity, such as *refactoring*, *test-first programming*, and indeed *pair programming*. These are all currently receiving most attention as elements of EXtreme Programming (XP) (see for example [1]), although all are pre-existing techniques. It is not a coincidence that XP's values include both a thorough-going concern for efficient working and an insistence that developers should be trusted to make decisions about development. Similarly, developers will only be able to make appropriate use of XMI if they are trusted to make the judgements about whether this is an appropriate, efficient and reliable way to get their job done in a given instance. The ability to make such judgements has of course to be learned, and mistakes may be made. It is the author's opinion that "grass roots learning" of this kind has an important role to play in the development of software engineering as a discipline, symbiotic with the roles played by people who research and develop tools, techniques and and processes.

5.2 Skills required of the developer

The skills required of the developer of minitools like those described here fall into four main categories, as follows.

Understanding of the UML metamodel The developer needs to understand enough about the structure of UML models to be able to find the information of interest. Most developers will not initially have this understanding: one can use UML without understanding the architecture of the language, and most books and courses on UML do not emphasise the metamodel. However, it seems to be rather easy to acquire the level of understanding required; for example, a new member of the author's team was proficient in a few days. A useful technique is to build an extremely simple "test" UML model which incorporates just the features which are of interest, save it as XMI and study the resulting simple XMI file.

Understanding of XML and proficiency with XML tools The developed needs to be able to read an XMI file and manipulate it with appropriate XML tools. This kind of skill is increasingly widespread as it is required in many other areas of development. It also helps that there is a wide choice of

XML tools; the developer of minitools can use whichever s/he is comfortable with.

Proficiency with a suitable language Scripting languages like Perl and Python are widely used in development for many purposes. It is also possible to use any other general purpose programming language to build minitools, provided only that appropriate XML tools are available. (Language choice is discussed further in the next subsection.) Thus language knowledge is not likely to be a barrier.

Confidence with rapid tool development There is a level of experience and self-confidence needed for a developer to be comfortable deciding when it is worthwhile to develop a minitool and rapidly doing such a development. Part of what is required is familiarity with tools and procedures for version control, testing and debugging, without which it is not possible to be confident in one's results. The developer also needs to have enough experience of their own development speed, since otherwise it is difficult to estimate the cost-benefit of a minitool development.

5.3 Choice of languages and language tools

Perl, as used in this paper, is a natural choice for tasks of this kind because it is already widely used by the developers on whom we focus, has good text manipulation facilities, powerful data structures and a wide range of modules including XML parsers and other tools. Other languages might, however, be more appropriate for a particular developer. Python is a particularly interesting example, sharing many of the advantages of Perl whilst being a simpler and arguably more elegant language. A recent technical report by Porres [13] describes an open-source Python toolkit developed specifically for manipulating UML models, which would be interesting to the Python developer undertaking mini-tool development. In [4] (the performance tool integration example mentioned above), Python was used because it was preferred by the person writing the scripts, although Porres' toolkit was not used as it was already simple to accomplish the task without it. We used a DOM-based parser in that instance, chiefly because this made it easy to modify the XMI file as well as analysing it.

There will also be cases where any general purpose programming language

is "over-kill" and XML tools are preferred. For example, XQuery may suffice to answer simple analysis questions. There are also circumstances under which the XML language XSLT is a natural choice, but this deserves a word of caution. XSLT is best suited to the re-presentation of one XML document as another, with minimal alteration to the structure of the document. It is not a general purpose programming language, and over-using its control constructs can easily lead to very poorly structured code. Moreover, naive use of XSLT can lead to very poor performance, which may be inadequate for even moderately sized UML models.

5.4 Kinds of tool integration

We have considered briefly how simple tool integration may be facilitated by the use of XMI files to transfer data between tools. It is important to recognise the deliberately limited ambitions of this form of tool integration. Brown et al. provide in [3] a wide ranging survey of approaches to tool integration which is useful although the book, which dates from 1994 and is now sadly out of print, predates UML and XMI. Drawing on [14] they describe four levels of tool integration:

1. data integration: tools are able to swap data, but their control flows may remain separate;

2. control integration: in addition to being able to swap data, tools are able to share control flow, e.g. by making direct requests of one another, but may use disparate presentation mechanisms;

3. presentation integration: in addition to the above, tools have consistent presentation metaphors and support similar mental models in the tool user;

4. process integration: in addition to the above, the tools support the development process in a consistent way, supporting a common set of assumptions about what is involved in carrying out a process step.

We have only been considering data integration, and this is the only kind of integration which XMI directly supports. To go beyond requires negotiation of considerable agreed infrastructure between tool providers, and history shows that this is hard to achieve: several previous attempts have

foundered after initial enthusiasm. We may speculate, however, that if the use of XMI for lightweight data integration of tools becomes widespread, demand for "deeper" tool integration may again grow, and might be achievable with the weight of bodies like the OMG behind it and if it were founded on an already existing base of users of XMI. Indeed, it may be argued that the OMG's vision of *model driven architecture* (`http://www.omg.org/mda/`) will not be practically achievable without such tool integration issues being carefully addressed, for it seems doubtful that developer organisations will allow their models to drive their development in the way that the OMG envisages if they cannot be confident of an effectively competitive market of well integrated tools to support them in doing so.

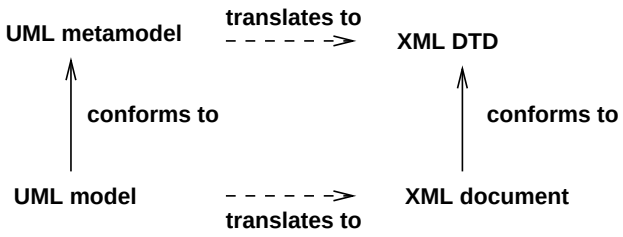
6 Technical problems and limitations

6.1 XMI incompatibilities

We have so far talked as though XMI laid down in full detail how to get from a UML model to an XML document. This is not so: XMI does both more and less than that. First, the XMI standard is not limited to use with UML; it gives rules for generating DTDs and documents from any MOF-compliant language. MOF, the OMG's meta object facility, can be considered a language for specifying languages; indeed for many practical purposes it can be seen as a small subset of UML itself. The basic idea is that for any MOF-compliant language a square like that shown in Figure 5 can be constructed, where the content of the "translates to" arrows is specified (loosely) in the XMI standard and the "conforms to" arrows are the standard ones of UML and XML respectively. (That is, an XML document conforms to a DTD if in the usual XML sense the XML document meets the specification described in the DTD, and a model conforms to a metamodel if it is a correct instance of the metamodel: for example, a correct UML1.3 model conforms to the UML1.3 metamodel.)

It is important to remember, however, that because the XML side of the diagram is much less expressive than the UML side, it is inevitable that information is lost in the translation. By no means every XML document which conforms to an appropriate UML DTD will be the translation of a conforming UML model, and this is not (solely) a fault of the current XMI standard but is inherent. (In fact, it would be possible to narrow the gap

Figure 5: How XMI transforms models and metamodels



and insist that DTDs should be much more prescriptive; and the migration to XML Schema will make this easier, and perhaps desirable.)

Further, because the translation rules in the XMI standard are not prescriptive, it is possible to construct different DTDs for the same MOF-based language, for example, different UML DTDs even for the same version of XMI and UML. For practical purposes we need to agree not just on a version of UML and of XMI, but even on a particular UML DTD.

The current formally adopted version of UML is 1.4 [6]; UML version 2.0 is under development. For UML1.1 an official DTD was included in the XMI specification, and for UML1.4 one was developed under the aegis of the UML revision task force (where it more naturally belongs). For UML1.3, however, there was no official DTD in either place, and problems have arisen from the widespread use of several.

Similarly, the XMI standard itself is under revision; for example, the current version of XMI at the time of writing is 1.2[9], but most tools use the previous version, 1.1 [5], and indeed an XMI1.1 DTD is provided with the latest version of UML, 1.4 [7]. Moreover whilst the versions of XMI currently available in tools use DTDs alone as the mechanism for specifying a family of XML documents, [8] takes advantage of the more expressive XML Schema language. This provides important technical advantages; for example, more ill-formed UML models can easily be eliminated, and there are gains in elegance, conciseness and readability. It seems unlikely, however, that DTDs will be abandoned for some time.

Naturally, many tools currently lag behind the current standards, often more than is apparent to the casual user. For example, in experimenting with one popular UML tool[2] I discovered that although it appears to the tool user

[2]naming which seems unfair, as I have no reason to believe that many tools do not do

that the version of UML used is rather consistently UML1.3, the same is not true "under the hood". A UML1.1 DTD was in use and, for example, saving a model involving a `<<include>>` dependency between use cases resulted in an XMI file which recorded not a dependency but a generalization between the use cases. That is, the stored form used the UML1.1 structure, and the tool internally translated between versions.

From the point of view of the developer, the need to be aware of these version issues both for UML and for XMI is a nuisance. However, it is to be expected at this comparatively early stage of both technologies and the signs for convergence appear good.

6.2 Graphical information and modifying models

Currently, a major limitation of the XMI standard is that it does not include a standard way to save layout information. For example, though we have standard ways to record that the user's UML model includes classes A, B and C, we do not have standard ways to record how big the user made the rectangles representing those classes and in what positions they were placed in the class diagram. A naive view of what a design model is suggests that this information should not need to be stored; for many purposes, such as verification and code generation, layout information is irrelevant. However, to use a language effectively requires mastery of not only its syntax and semantics, but also of its *pragmatics*; this includes, for example, unarticulated conventions about how the language is used in practice. Layout is important in the pragmatics of UML. For example, conceptually related model elements will be drawn close together in diagrams (whether or not this is dictated by minimising crossing lines etc.), blank space will be left in diagrams for elements which the designer thinks may be needed in future, conceptually similar relationships will probably be represented with parallel lines, etc.

In summary, users care about the pragmatics of their models, so they will not readily accept a tool which ignores their own diagram layout and imposes its own; for example, it would be unacceptable if when users save and reload a model its layout changed. Therefore tool developers find ways to record graphical information alongside the semantic information recorded in standard form in XMI. For example, they may use XMI's extension section which is intended for tool specific information, or they may save graphical

---

the same "trick"

information in a separate file. The existence of several different mechanisms hampers the interchange of UML models between tools. As part of the development of UML2.0, however, a standard diagram interchange format is being developed. Progress can be tracked at `http://www.omg.org/techprocess/meetings/schedule/UML_2.0_Diagram_Interchange_RFP.html`. It seems almost certain that diagram interchange will be incorporated into XMI via some kind of diagram metamodel.

These aspects of UML are inherently informal, and therefore difficult to make use of in model analysis and manipulation tools. For this reason the absence of graphical information in standard form from XMI has, in many cases, little impact. Where it is important, however, is in situations where there is a need for a tool to modify or create a model, including its graphical information. For example, there are many circumstances in which it would be useful for an external tool to generate a sequence diagram to demonstrate the possibility of a particular interaction. Until the diagram interchange standard is agreed, there is currently no standard way to record the layout of such a sequence diagram in XMI. However, less ambitious model alterations like modifying names of elements generally work now and will sometimes suffice.

6.3 Very large UML models

The little examples we have dealt with so far have been suitable for working with small UML models; but, as the XML parser builds in memory a tree corresponding to the whole XML document, they are not suitable for use with the kind of huge models sometimes encountered. An advantage of the ubiquity of XML is that similar problems are encountered in many other fields and various solutions are available. For example, *event-driven parsers* (such as those based on SAX) parse the file on the fly and invoke user-provided functions or methods when particular constructs are encountered. In practice, the major problem seems to be the time and memory taken to save and load such models in UML tools, rather than any particular difficulty in processing them.

7 Conclusion

We have demonstrated that the use of XMI can make it easy to carry out small tasks on (possibly large) UML models. In particular, such techniques can make tool integration available to developers in a way which is quite new. Whilst we acknowledge the danger that such rapidly written ad hoc scripts may contain bugs, in the context of tasks carried out by the developer who writes the script the risk is likely to be acceptably small, especially when the alternative is to carry out the same task by hand: carrying out repetitive, boring tasks on large models is also error prone! Further work will focus on the development of lightweight utilities, patterns and methods for maximising the benefits and minimising the risks of these techniques.

7.1 Acknowledgements

I would like to thank the referees, and the EPSRC for financial support in the form of an Advanced Research Fellowship. I am also grateful to Dale Mahan of Boeing for providing me with very large UML models with which to experiment.

References

[1] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[2] Steve Brodsky. XML metadata interchange. `http://www-4.ibm.com/software/ad/standards/xmi.html/xmiwhite0399.pdf`.

[3] Alan W. Brown, David J. Carney, Edwin J. Morris, Dennis B. Smith, and Paul F. Zarrella. *Principles of CASE Tool Integration.* Oxford University Press, 1994.

[4] C. Canevet, S. Gilmore, J. Hillston, and P. Stevens. Performance modelling with UML and stochastic process algebras. To appear in the proceedings of UK PEW 2002, May 2002.

[5] Object Management Group. XML metadata interchange, version 1.1, 2000. Available from http://www.omg.org as document formal/00-11-02.

[6] Object Management Group. Unified modeling language, version 1.4, 2001. Available from http://www.omg.org as document formal/2001-09-67.

[7] Object Management Group. XMI 1.1 DTD for UML 1.4, 2001. Available from http://www.omg.org as document ad/01-02-16.

[8] Object Management Group. XMI production of XML schema final adopted specification, 2001. Available from http://www.omg.org as document ptc/01-12-03.

[9] Object Management Group. XML metadata interchange, version 1.2, 2002. Available from http://www.omg.org as document formal/02-01-01.

[10] Ivar Jacobson, Grady Booch, and James Rumbaugh. *Unified Software Development Process*. Addison-Wesley, 1999.

[11] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: A consistency checking and smart link generation service. *ACM Transactions on Internet Technology (TOIT)*. to appear.

[12] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Andrea Zisman. BOX: Browsing objects in XML. *Software Practice and Experience*, 30(15):1661–1676, December 2000.

[13] Ivan Porres. A toolkit for manipulating UML models. Technical Report TR441, Turku Centre for Computer Science, January 2002. Available from `http://www.abo.fi/~iporres/Projects/fog0000000023.html`.

[14] I. Thomas and B. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, 9(3):29–35, March 1992.