Steven Kelly, Matti Rossi, Juha-Pekka Tolvanen

What is Needed in a MetaCASE Environment?

In this paper we look at ways of effectively implementing software development environments through metaCASE tools. MetaCASE tools offer fast and economical means of supporting tailored or homegrown systems development methods, yet they have not been taken into use widely due to their perceived complexity and the lack of development process maturity in most organisations. We offer a list of generic requirements for these tools and demonstrate their use through evaluating the MetaEdit+ tool against these requirements. The requirements are gathered from existing literature on method engineering.

1 Introduction

Customisation of software is not a new idea: many application domains, such as ERP software or telecom switches, apply it. Having a customised tool makes users more productive, shortens learning curves, and reduces errors. In software engineering, customisation of modelling and code generation tools is not done extensively due to the high costs and expertise required. Having metamodelling facilities, e.g. MOF or its Eclipse implementation EMF, and frameworks for CASE-style graphical editors, e.g. Eclipse's GEF, does help, but still the costs can be prohibitive. For example, with Eclipse it takes about 5000 man-days, roughly 25 man-years, to implement support for UML [Strö05].

A true metaCASE environment can offer major reductions in these costs. For instance, with a metaCASE environment it takes less than 5 man-days to implement support for UML; for another metamodel, the difference was a factor of 2000 [Kell04]. Perhaps the defining feature of such a metaCASE environment is that the user should merely *specify* the desired modelling language, without having to program either it or any tool functionality to support it.

In this paper we look at the functionality of a metaCASE environment that goes beyond plain metamodelling into tool construction, modelling language evolution, model and metamodel sharing etc. As such, it extends current research on specifying evaluation criteria and comparing metaCASE tools [MRTL93; MaHR96; IsLa97]. In the next section we look at general requirements for tools for modelling language definition and use. The third section applies these to MetaEdit+ environment to demonstrate their application. In the last section we provide conclusions and future research.

2 Requirements for MetaCASE Enviroments

Many researchers and practitioners have presented wish lists for metaCASE environments. The literature also uses terms such as method engineering tools, CAME (Computer Aided Method Engineering) tools, CASE-shells, metamodelling tools or meta tools to denote environments where modelling support can be defined by the user. Some of these only handle either definition of new modelling languages or support for modelling in those languages; a metaCASE environment includes both.

We surveyed the literature on tool proposals and tool comparisons to identify common requirements for metaCASE environments. As a basis of our synthesis we have used a few key articles from previous metaCASE research [KoKo84; SoTM88; KuWe92; MLR+92; KaSc93; MRTL93]. These were among the first articles to enumerate specific tool requirement or capability lists. As pointed out in Leppänen's recent survey of the area [Lepp05], these articles still present the state-of-the-art in the field.

2.1 Definition of modelling languages

The first obvious requirement is that a metaCASE environment can specify the concepts, rules, and symbols of individual modelling languages as well as their interconnection rules. Moreover, the modelling language definitions should be as complete as possible, and should be relatively easy and fast to make, as far away from programming a CASE tool as possible.

The cornerstone when defining a modelling language into a CASE environment is the definition language i.e. metamodelling language [TMHY80]. A powerful metamodelling language guarantees that a modelling language can be successfully defined, but trying to support all possible cases can lead to the language can become very time consuming and complicated to use for most cases. The expressive power of the metamodel should thus be maximized, but without introducing undue complexity: language definition must be efficient [Klin93].

The simplicity and ease of use [KoKo84; SoTM88] of the metamodel definition facilities are vital. The system is intended to provide a platform for developing CASE tools and the tool developers are assumed to be experts in the domain of the modelling language, not database or tool implementation experts [KuWe92]. To speed up modelling language development, there should be frameworks or "starter kits" with reuse support tools. The system should be able to catch and flag common errors in environment definition.

2.2 Metamodelling process and metamodel management

An important part of ease of use is being able to see the results of actions immediately. This calls for the possibility to incrementally test parts of the metamodel implemented so far [SoTM88; Ka Sc93]. This can be seen as prototyping of modelling languages, where parts of the modelling language can be tested and refined while developing the overall metamodel.

Support for incremental metamodelling also requires that the models made with previous versions of the modelling language are automatically updated to reflect the changes, whenever possible [KeTa94]. Modelling languages, and especially their usage, usually evolve as time goes by and there is thus a corresponding need to change their definitions in the tool. Experience shows that the most common changes are the addition of new metamodel elements and the removal or deprecation of old metamodel elements. Also, rules are more often relaxed than tightened, whereas the changeability of symbols appears to vary widely from case to case.

A metaCASE environment should provide functionality for metamodel management similar to a CASE tool's functions for model management. This includes browsers, documentation tools, libraries for metamodels, and setting of access rights for editing metamodels.

2.3 Creation of modelling tools

Based on the modelling language definitions, a metaCASE environment should provide the necessary modelling tools to support systems design tasks with the given language. These include different kinds of editors, toolbars, dialogs, online help, etc. The creation of these tools should be automatic, based on the metamodels. Although many current modelling tools focus on creating graphical editors, also other types of model representations should be supported, like matrixes, tables, text etc [KeLR96].

Whilst one way to create the tools is to generate the necessary code, in general it is better that the environment already includes generic tools, and these configure themselves based on the metamodel. This makes language development and maintenance safer and easier for the method engineer. This approach allows the separation of parts that change infrequently (base tool behaviour such as object selection and movement) from parts that change more frequently (the metamodel and its symbols). The same separation is also found when considering the different core competences of the metaCASE tool provider and the metamodeller, or the commonalities and variabilities between different modelling tools made with the same metaCASE tool.

2.4 Repository

Model data differs from traditional single user applications in that there are often multiple users who need to interact simultaneously, and from traditional multi-user database applications in that the data elements may be reused at a fine level of granularity to form a complex network.

Both models and metamodels should be stored in a multi-user repository and be accessible to developers. This offers an effective way to share the metamodels and update them during modelling language evolution: a customised tool must evolve.

Vessey and Sravanapudi [VeSr95] provide an extensive set of references and motivation on the requirements for multi-user CASE. They divide the needed functionality into taskware (basic CASE

functionality, no communication necessary), teamware (CASE information sharing, access control and monitoring), and groupware (non-CASE communication, time and meeting management). We agree with them that the most prominent needs are for teamware, in particular the ability to share information, with concurrency control 'to resolve conflict and support tightly coupled group activities'. They perceive groups as working most frequently in an asynchronous mode, but also sometimes needing to access shared resources at the same time.

The ability to identify not only the models, but also their components such as individual objects, and possibly even individual properties, is important for later reuse of parts of the models and for effective manipulation of models [KoKo84]. The resulting highly interlinked network has been seen as a major departure from relational models in the repository support of the CASE tool [SoTM88]. Typically, the data in CASE repository is made out of small objects, which have complex dependencies and internal structure, so their efficient management becomes a focal issue.

2.5 Code generation and reports

In addition to the model editing and storing, a metaCASE environment should allow the definition of code generators, various model analyses, and model documentation reports. Reports to check models are also needed: although a good metamodel includes all the rules of the modelling language, checking them cannot be fully automatic: some rules should be checked only after models are considered complete.

2.6 Several levels of modifiability

As there are multiple roles associated with modelling language and IS development, there are several different views on the needed modifiability [KaSc93; MRTL93]. At the organizational level there is a need to develop a common language, or a reference model, for ISD, whereas at the individual project level there are contingencies that force the users to adapt the modelling language to the situation at hand. At the user level there are usually individual preferences about the way of interacting with the CASE tool.

2.7 Interchange format for metamodel and model definitions

The resulting CASE tool should provide importing and exporting of both models and metamodels. Importing should be incremental: previously imported data from the same exporter should be updated automatically, rather than creating duplicates.

Part of the reason for an interchange format is to support multiple users working in the same tool. In tools without a multi-user repository, this will normally be the main file format, although support for sharing smaller collections of model elements is also useful. The other reason for an interchange format, at least in theory, is to allow data interchange with different modelling tools or other tools. The history of attempts at such interchange formats is not encouraging. The suggested standards have generally been poor even on paper, implementations supporting them rare, and the chance of finding two different tools between which interchange works negligible.

The most recent attempt at an interchange format is XMI. The OMG has XMI versions 1.0, 1.1, 1.2 and 2.0, with 2.1 under development. According to Google at the time of writing there are 865 XMI files on the web using version 1.0, 78 for 1.1, 64 for 1.2, and 34 for 2.0 (released in 2003). Those figures give some indication of the adoption of XMI as a format, and discussions with researchers bear out the negative impression. It seems everybody was interested in XMI when it first came out, but most who actually tried to use it found it lacking. Subsequent versions have certainly not improved the situation.

An interchange format between modelling tools should in any case only be used as a one-shot transformation: trying to maintain the same models or metamodels in two tools is generally not a good idea.

3 Solutions in MetaEdit+

MetaEdit+ is a customizable CASE environment that supports both CASE and metaCASE functionality for multiple users within the same environment. It supports and integrates multiple modelling languages and includes multiple editing tools for diagrams, matrices and tables. It was developed in the MetaPHOR project, which had earlier developed the single user MetaEdit metaCASE tool [KeLR96]. Figure 1 shows the architecture of MetaEdit+, which is a client-server with the server containing the repository and a thin server process to communicate with clients and handle locking etc. Each client containing a central MetaEngine and various tools, through each of which a user can view and edit design objects in a particular way. The MetaEngine is a service layer which presents a public interface to the models and metamodels in the repository. Software tools request services of the MetaEngine in

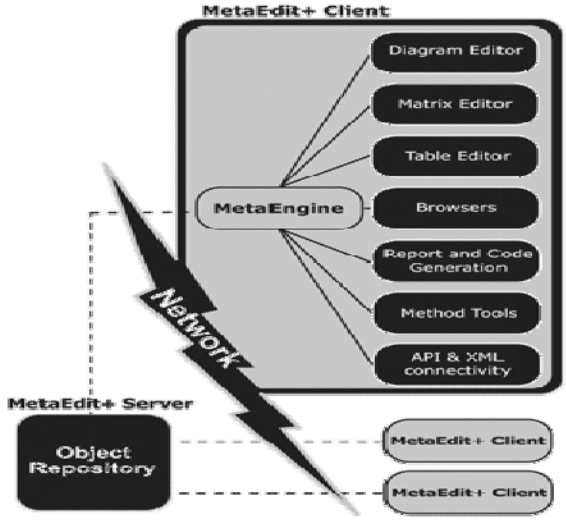

Figure 1: MetaEdit+ architecture

accessing and manipulating repository data. This design choice allows flexible integration of new tools, each only responsible for its own view (including operations) on the same underlying repository data. A tool can be for example a diagram, matrix, or table editor or report generator.

The adoption of full object orientation enables flexible organization and reuse of software components in the environment and a high level of interoperability between tools. MetaEdit+ supports Data independence as defined in traditional data base theory i.e. tools operate on design information without "knowledge" of its physical organization, or logical access structure. Representation independence forms a continuum with data independence and it allows conceptual design objects to exist independently of their alternative representations as text, matrix or graphical representations [SLTM91]. This principle allows flexible addition of new tools, each one only responsible for its own paradigmatically different view on the same underlying data.

In the rest of this section, we answer the questions raised in Section 2. The following subsections are organized as answers to the questions raised above.

3.1 Definition of modelling languages

The core constructs of MetaEdit+ are in its conceptual meta-metamodel called GOPRR [SLTM91, KeLR96]. The top-level GOPRR concept is the Graph. A Graph can contain Objects, which are linked together via bindings. The centre of each binding is a Relationship and it may have two or more Roles, allowing n-ary relationships. One Role leads out to each Object involved, either connecting directly to it or via a Port on the Object. A Graph can also specify explosions from each Object, Role or Relationship to possibly multiple Graphs, and each Object can also specify a single decomposition sub-Graph.

All of these concepts can have Properties, whose values can be simple (string, number, Boolean, text etc.) or complex: references to another concept or collection of concepts. This allows arbitrarily deep nesting and complex networks of objects, e.g. a UML-like 'Class' object could have an 'Attributes' property containing a collection of 'Attribute' objects, each specifying strings 'Name' and 'Data type', a Boolean 'Derived?' etc. String properties can also be restricted to be from a list, e.g. an Attribute could have a string property, 'Visibility', which could only take values of "public", "private" or "protected".

Metamodels are defined through form-based tools, one for each concept. Figure 2 shows the definition of UML's Attribute in an Object tool, and of its Visibility property in a Property tool.

Most rules and constraints in metamodels are to be found in the definition of Graph types. In GOPRR, an Object type does not specify which Relationship types it may take part in, nor vice versa. This is not a feature of an Object itself, but of a particular Graph type where that Object is used: specifying it in the Object type would severely restrict the ability to reuse the same Object type in different Graph types. A Graph type thus specifies the legal bindings of Objects, Relationships, Roles and Ports, including the cardinality of the Roles: how many times may a given Role repeat within a single binding, e.g. an inheritance relationship may have only one

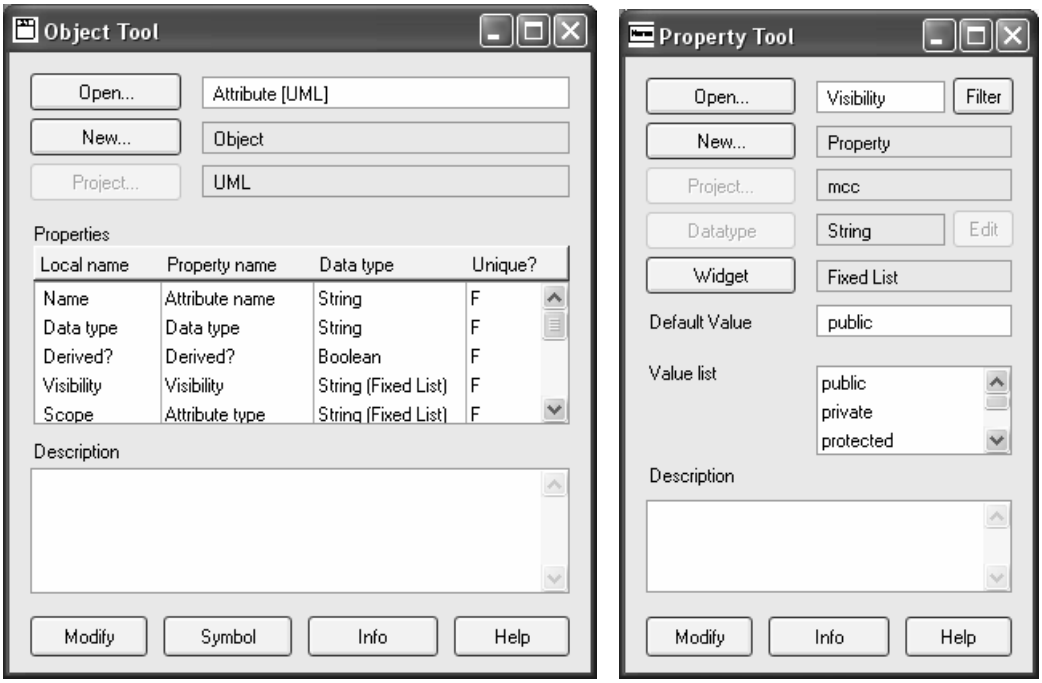

Figure 2: Metamodelling tools for UML Attribute and Access level

Generalization role to the superclass, but the Specialization role may occur 1..N times. Bindings thus handle the expression of most of the rules about how objects may be connected together, expressing the rules in a simple format that is easier to understand and manipulate than other approaches such as set theory, predicate logic or scripting languages like OCL.

In addition to bindings, other constraints can be expressed, including connectivity and port constraints. Figure 3 shows the constraints list for a UML State diagram and the form for specifying the highlighted connectivity constraint. It also shows a form specifying a Port constraint for an example metamodel of electrical circuits: each object there has ports that specify their voltage, direction (in or out), and type (analogue or digital), and the constraint thus specifies that only ports specifying the same voltage can be connected. Similar constraints for different direction and the same type would also be specified. Again, the specification of the constraints is simple and requires no programming. The set of possible constraints is based on those that are found in real modelling languages, both standard and domain-specific. Whilst this cannot of course cover all constraints it is possible to invent, our experience is that those provided have been sufficient. Indeed, frequently metamodellers are initially enthusiastic about creating constraints, but feedback from modellers soon convinces them to allow richer models, and extend code generation to supply the required semantics.

The GOPRR data model makes a distinction between the representational and the conceptual aspects of a modelling language to allow for multiple different representations of the same concept. This approach allows the CASE environment to support a wide range of modelling languages and visualisations. The GOPRR model is object-oriented. It includes both abstract and concrete inheritance of structure and behaviour, polymorphism, overloading, and class/object paradigm. The true object oriented nature of the design and implementation of GOPRR allows fine-grained identification of the units (which we call components), as both the types and instances can be identified into the property level regardless of their values. The Graph and Project, as well as Property's ability to contain other types allow versatile support for modelling complex objects and recursive structures.

MetaEdit+ is designed from its information model up to provide strong support for reuse. All GOPRR components can be reused, on both type and instance levels. In particular, graphs display a type-free interface to components, allowing them to be reused across different modelling languages, but still supporting the linking of interface relationships of an object in a higher level graph to the objects within

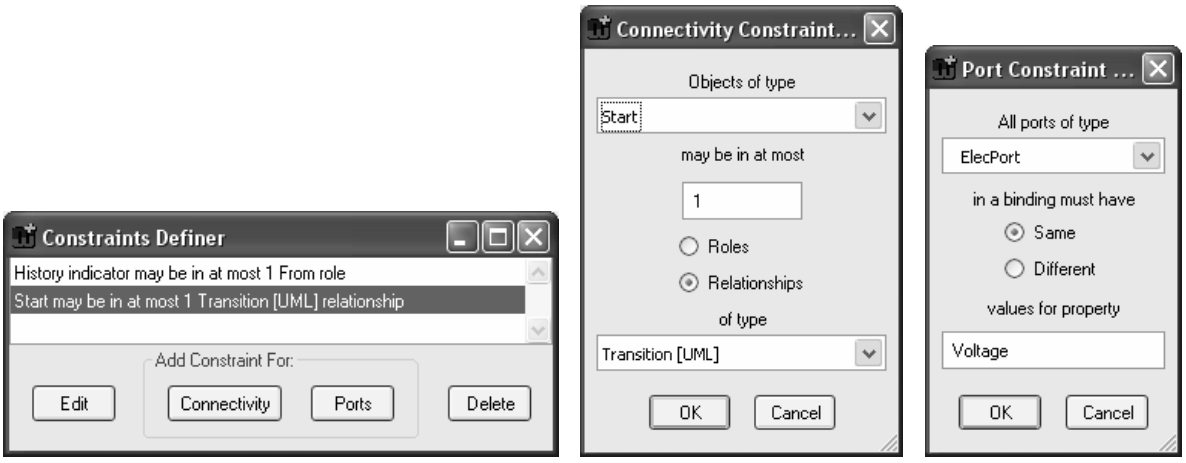

Figure 3: Constraints list, Connectivity and Port Constraint Definers

the lower level decomposition graph. This allows graphs to be reused in a similar way to components in CAD, including both black-box and white-box reuse.

3.2 Metamodelling process and metamodel management

As MetaEdit+ allows incremental specification of modelling languages, it must allow for incomplete metamodels, while allowing users to model by using these partial specifications. Method engineers can change components of a metamodel even while system developers are working with older versions of the metamodel. The modelling language can be developed and simultaneously tested on the method engineer's workstation in much the same way as described in [Hedi92]. As the method engineer commits his changes to the database, the other users' models update to the new modelling language specification (see [KeTa94; KeLR96] for discussion about the locking implementation, which is critical for this kind of modifiability to work).

Data continuity, i.e. that existing models remain usable even after metamodel changes, is confirmed by a number of checkings and limitations to the metamodel evolution possibilities. The idea is that the user can always be guaranteed data continuity while working with partial metamodels. In cases where old descriptions are in conflict with new modelling language definitions, the old data remains intact. For example if types are removed from a metamodel, the models can still have instances of the deleted types, but there is no possibility to add new instances of the deleted types. The old models are automatically updated to refer to the new metamodel version at the next transaction boundary after the modification.

Metamodels can be managed with a variety of tools showing individual metamodel elements, browsers showing trees and lists of how the elements are related, and management tools for importing, exporting and deleting metamodel elements. The management tools all work at a high level, allowing users to choose Graph types to operate on, rather than having to select the individual types that will be exported or deleted. This prevents the user from exporting a Graph type but missing a Property type needed in one of its Object types, or conversely stops him deleting a Property type that is still used by some Object type. All these tools also allow the user to see which types are used by the selected type, and which types use the selected type.

3.3 Creation of modelling tools

MetaEdit+ includes a comprehensive set of generic modelling tools which adapt themselves to the metamodel currently being used. The tools' behavior, menus, toolbars and dialogs all change to reflect the metamodel, without any work on the part of the metamodeler. This allows the metamodeler to concentrate on the metamodel, not its implementation in code, and provides the largest savings in the costs of building a new modelling language and tool support for it.

Tool support for reuse is built into the MetaEngine, and is thus available in all editors, browsers etc. This includes the ability to select graphs, objects, relationships, roles and properties for reuse, selecting them based on their type or via another component that already uses them. In addition,

browsers offer wildcard string-based queries against type and identifying property.

3.4 Repository

The heart of the MetaEdit+ environment is the Object Repository. The repository is implemented as a database running in a central server: clients communicate only through shared data and state in the server. All information in MetaEdit+ is stored in the Object Repository, including metamodels, diagrams, matrices, objects, properties, and even font selections. Hence, modification of models or modelling languages in one MetaEdit+ client is automatically reflected to other clients on transaction boundaries, guaranteeing consistent and up-to-date information.

The Object Repository itself is designed to be mostly invisible to users, allowing collaborative teamwork with the minimum of distractions. The use of the repository is visible only when a user starts or exits MetaEdit+, opens or closes projects, and commits or abandons transactions. A repository is composed of projects, each of which contains a set of graphs that describe a particular system, and possibly some metamodels. Opening a project reads all the models in that project and their top level objects, so they are visible to users e.g. in browsers. However, not all fine-grained components (i.e. individual sub-objects, properties etc.) are read: these are only read as they are needed, e.g. when they are being displayed in a model which the user opens. If all data were loaded immediately, start-up times would be too long for large repositories: in some projects MetaEdit+ is used in there are hundreds of users and tens of gigabytes of models. Loading incrementally in this way has been found to provide a good compromise: the initial start-up time is a few or several seconds, and loading a cluster of data subsequently – e.g. all the fine-grained components needed when opening a model for the first time – takes less than a second.

3.5 Code generation and reports

Code generation, documentation generation and model-checking reports are all performed in MetaEdit+ by running reports. Reports access information in the repository and transform it into various text-based outputs. Reports can also output information in various graphical formats, call subreports, query information from the user with a dialog, or call external programs and commands.

MetaEdit+ includes a number of generic reports that will work with any metamodel, such as generating documentation in HTML, RTF or Word formats, or performing elementary checks on models. The library of existing metamodels that accompanies MetaEdit+ also includes appropriate code generators, e.g. for SQL from ER diagrams or for C++, Smalltalk, Java and other object-oriented languages from Class Diagrams and similar metamodels. With the Report Browser users can view and edit these, and most importantly make their own new reports and queries on the repository.

The MetaEdit+ reporting language is a domain-specific language, designed specifically for the task of transforming the object structure of a model into text. Whilst existing languages were considered, none seemed to fit the task well: there were languages for processing one text stream into another text stream (e.g. Perl), or for processing one object structure into another (e.g. any object-oriented language), but not for navigating an object structure and outputting text.

The syntax is vaguely C-like (curly brackets and semicolons), with common keywords based on pseudo code (if..then..else..endif, dowhile, foreach etc.). Two key areas in the language are its support for navigation around the model structures and its extensive use of streams. All output from report commands goes to the current default output stream: for instance, the simplest command is any single quoted string, which is copied to the output stream (e.g. line 1 in Listing 1). Loop structures combine the normal control function of a loop with model navigation, e.g. line 2 below will run the block of lines 3–11 once for each State object in the current model: we can consider that we navigate so that we are ‘in’ the State, run the block, navigate to another State and so on. Thus references to properties like :Name in line 4 will refer to the Name property of the then current State.

Line 7 shows another example of navigation: "do ~From~To.State". Starting from the outer loop's current State, it says to crawl along any From role and its To role into the next State. That one line replaces twelve lines that would be necessary if C# would have been used as the reporting language. This kind of pattern is very common in any code generation or reporting on a model. Users of other tools whose reporting languages are standard 3GLs will thus quickly find their code full of similar blocks of 12 lines of code: ironic in tools intended to save developers from such unproductive code duplication.

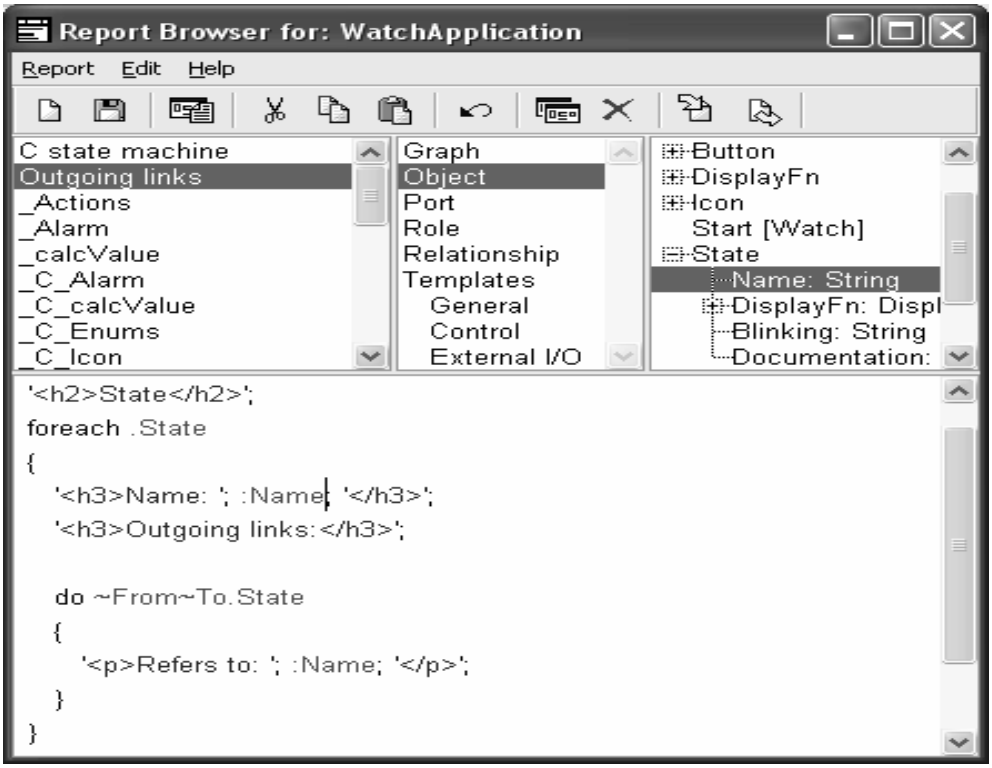

Figure 4: The Report Browser with its report and concept lists

```
1    '<h2>States</h2>';
2    foreach .State
3    {
4       '<h3>State Name: '; :Name; '</h3>';
5       '<h4>Outgoing links:</h4>';
6
7       do ~From~To.State
8       {
9          '<p>Refers to: '; :Name; '</p>';
10      }
11   }
```

Listing 1: Report to generate a list of States and their successors

In the report above, the only outputs were of fixed strings or property values, and all went to the default output stream: a text window opened after the report has run. Whilst that is useful for checking reports, often we want to have the output going to a file. For instance, we could enclose the lines above inside a filename; … write; … close; structure as in Listing 2:

```
0    filename; :ModelName; '.html'; write
        ... lines 1-11 above ...
12   close;
```

Listing 2: Sending output to a file named after the graph's ModelName property

The first thing to note here is that all of the HTML from Listing 1 will now be output to a file called <ModelName>.html, where <ModelName> is the name property of the model the report is run on. In other words, the output stream for lines 1–11 has been redirected to a file. Interestingly, the same approach is used in line 0 to form the name of the file. The "filename" command opens a new, temporary stream. :ModelName writes the name of the model on that stream, and '.html' writes those five characters on the stream, together forming the file name. Then the "write" command closes that temporary stream, reads its contents, creates a file of that name, and redirects subsequent output into that file. This use of streams has also been found to be appropriate when building command strings to run in the operating system shell, e.g. to compile the generated files, and when building the name of a subreport to call, e.g. calling a different subreport for each type of object found. The latter saves writing "if type = 'foo' then subreport; 'foo'; run; endif; if type = 'bar' then subreport; 'bar'; run; endif;", replacing it with just "subreport; type; run;" – the "type" command outputs the name of the type of the current element, which is then taken as the name of the subreport to call. This can of course be

extended arbitrarily to allow the reporting language equivalent of double dispatch or multimethods.

As well as providing a domain-specific language for the task of generating code and reports from models, MetaEdit+ offers a directed editor for editing reports (Figure 4). The list on the left shows the reports defined for this Graph type, whilst on the right is a list of the types present in the modelling language. The selection in the middle list determines whether the right hand list shows object types, relationship types etc., or then various command templates of the reporting language should as if…then…else…endif. Double-clicking an entry on the right inserts the selected type, property or template, allowing reports to be built easily even for new users. In the current development version there is also a full debugger that allows users to step through reports and follow the stacks of objects, outputs and subreport calls.

3.6 Several levels of modifiability

MetaEdit+ supports several levels of modification of the modelling language definitions based on GOPRR types. The core types of a modelling language are defined at the repository level (e.g. its GOPRR types). This can been seen as the development of a domain ontology [JPW+98]. At the metamodel level we can define how these components look to the user, (e.g. definition of the dialogs and symbols, which are used for inserting the concept instances into the repository). At the user level sub-views can be defined to support individual looks of the components and own styles of interaction (for example, systems analysts can just look at the high level descriptions of attributes and system developers can see the implementation details as well).

The repository definitions form the base on which the other modifications are built, while data in the repository remains consistent with the repository schema. Even if users have defined different sub-views on the models, or use different tools to access and modify the data, consistency is guaranteed.

3.7 Interchange format for metamodel and model definitions

Because MetaEdit+ uses a multi-user repository, there is less need for an interchange format between MetaEdit+ users. However, for the benefit of the single user version, MetaEdit+ offers a binary import/export file format for metamodels and/or models. Importing is incremental: previously imported data from the same exporter is updated automatically, rather than creating duplicates. Also, new data from the same exporter is linked in to previously imported data from that exporter, maintaining model consistency and reuse.

XMI suffers from the same problems as MOF, e.g. no support for n-ary relationships and too much dependence on UML. For those reasons it was not sufficiently powerful or flexible for use as the import/export format for MetaEdit+. Instead, we used GXL [Wint02]. GXL is significantly better than XMI in both architecture and schema details, and is supported by an broad mix of tools. Unfortunately, it is not yet supported by other major vendors: for their purposes it seems to have been thought wiser to quote XMI compliance and provide an XMI implementation that is incompatible with other vendors.

As MetaEdit+ 4.0 added the concept of Port, which is not yet present in GXL, we are currently forced to use our own extended version of GXL – further proving the difficulty of achieving a working interchange format. At the end of the day, though, all XMI and GXL versions are simply XML documents containing sufficient information describing models. As such, there are two ways to get data from XMI into MetaEdit+. One way is by transforming the XMI file into the MetaEdit+ GXL format, e.g. using XSLT. The other way is to have a program read the XMI file and make calls into MetaEdit+ to create corresponding objects etc. The MetaEdit+ API uses SOAP as its protocol, and so can be easily called from almost any client OS or programming language.

4 Conclusions and future research

We have demonstrated that a metaCASE environment can address the call put forth by the previous research. The environment provides a simple yet powerful meta-metamodel and advanced tools to develop and modify methods. The metamodelling tools guide the method developers in their tasks and immediately deliver a running environment for the modelling language. The incremental definition of modelling languages and support for data continuity allow an evolutionary approach to modelling language development. The first version of the modelling language can be defined quickly, then incrementally experimented with and modified at will until a satisfactory solution has been found. In the ultimate case, the system developers can to a certain degree modify or extend the modelling language while they work and thus tailor it to the task at hand, while preserving the integrity of the data with the other team members.

To summarize, the key architectural and implementation principles behind MetaEdit+ environment and tools are:

- the ease of use of the modelling language definition languages and tools,
- an integrated environment for modelling language definition and use,
- incremental development and testing of the new method components, and
- support for reuse at both type and instance levels

The MetaEdit+ environment has been successfully used for developing support environments for over 100 modelling languages, with commercial customers in twenty countries. We claim that the environment now provides most of the functionality expected from a full-blown modelling tool, while still supporting flexible and easy to use modelling language modification. With the support of a reusable library of textbook modelling languages, there is a possibility for supporting local needs and new innovative systems development practices with a modest effort. Most importantly, creating new modelling languages does not require specialized MetaEdit+ consultants to implement the modelling language into their proprietary tool. Researchers and innovative users can create variants of existing methods and combine available methods in new ways. As we expect to see more revolutionary and "standard" methods appear, this can be a cost effective and a low risk way to test and support them.

In the future, our aim is to carry out further empirical studies on the use of metaCASE environments in practice, as in the study of 23 cases by [LuKT04]. Another research area is at the tool level: we need to investigate effective means of cataloguing and searching modelling language components, i.e. to provide more comprehensive tools to support reuse in modelling language development. We have started an effort to develop a categorization framework and the development of advanced retrieval tools for modelling language components [Zhan00].

In 1995 we concluded a paper on requirements for metaCASE tools as follows: "The future CASE environment, in our opinion, can be described better as an evolving organizational knowledge base (design information system) rather than a passive data store for system descriptions. This implies that future environments must have a set of tools to handle the elicitation of ISD specifications and to guide the users in gathering information about the IS, as well as tools to co-ordinate their action during the development processes. The environment should also offer a seamless integration of the development steps and different types of tools. Finally, the environment should offer users enough flexibility so that when they demand changes, the environment can easily accommodate these changes" [MLR+95]. We see this work as taking steps towards that direction.

References

[Hedi92] Hedin, G: Incremental Semantic Analysis. Department of Computer Sciences. Lund, Lund University, 1992.

[IsLa97] Isazadeh, H.; D. Lamb: A Comparative Review of MetaCASE Tools. In: Systems Development Methods for the Next Century. Wojtkowski, Plenum Press, 1997.

[JPW+98] Jarke, M.; K. Pohl; K. Weidenhaupt; K. Lyytinen; P. Marttiin; J.-P. Tolvanen; M. Papazoglou: Meta Modeling: A Formal Basis for Interoperability and Adaptability. In: Information Systems Interoperability. B. Krämer ; M. Papazoglou, John Wiley Research Science Press**,** 1998, pp. 229-263.

[KaSc93] Karrer, A.; W. Scacchi: Meta-Environments for Software Production. In: International Journal of Software Engineering and Data Engineering 3 (1993) 1, pp. 139-162.

[Kell04] Kelly, S.: Tools for Domain-Specific Modeling. In: Dr.Dobb's journal, September, 2004.

[KeLR96] Kelly, S.; K. Lyytinen; M. Rossi: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Advanced Information Systems Engineering, proceedings of the 8th International Conference CAISE'96. P. Constapoulos, J. Mylopoulos and Y. Vassiliou. Berlin, Springer-Verlag**:**, 1996, pp. 1-21.

[KeTa94] Kelly, S.; V.-P. Tahvanainen: Support for Incremental Method Engineering and MetaCASE. In: 5th Workshop on the Next Generation of CASE Tools, Enschede, the Netherlands, Universiteit Twente, 1994.

[Klin93] Klint, P.: A Meta-Environment for Generating Programming Environments. In: ACM Transactions on Software Engineering and Methodology 2 (1993), 2, pp. 176-201.

[KoKo84] Kottemann, J. E.; B. R. Konsynski. Dynamic Metasystems for Information Systems Development. Fifth International Conference on Information Systems, 1984.

[KuWe92] Kumar, K.; R. J. Welke. Methodology Engineering: A Proposal for Situation Specific Methodology Construction. Challenges and Strategies for Research in Systems Development. W. W. Kottermann and J. A. Senn. Washington, John Wiley & Sons**:** 257-269, 1992.

[Lepp05] Leppänen, M.: An Ontological Framework and a Methodical Skeleton for Method Engineering – A

Contextual Approach, Doctoral Thesis, Jyväskylä Studies in Computing 52, University of Jyväskylä, 2005.

[LuKT04] Luoma, J., S. Kelly; J.-P. Tolvanen: Defining Domain-Specific Modeling Languages: Collected Experiences, In: Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04), Computer Science and Information System Reports, Technical Reports, TR-33, University of Jyväskylä, Finland, 2004.

[MaHR96] Marttiin, P.; F. Harmsen; M. Rossi: Evaluation of Two CAME Environments Using a Functional Framework: Findings on Maestro II/Decamerone and MetaEdit+. In: Method Engineering, Principles of Method Construction and Support. S. Brinkkemper, K. Lyytinen and R. Welke (eds.). London, Chapman-Hall, 1996, pp. 63-86.

[MLR+92] Marttiin, P.; K. Lyytinen; M. Rossi; V.-P. Tahvanainen; J.-P. Tolvanen: Modeling Requirements for Future CASE: Issues and Implementation Considerations. In: 13th International Conference on Information Systems, Dallas, Texas, ACM Press, 1992.

[MLR+95] Marttiin, P.; K. Lyytinen; M. Rossi; V.-P. Tahvanainen; J.-P. Tolvanen: Modeling Requirements for Future CASE: Issues and Implementation Considerations. In: Information Resources Management Journal 8(1995) 1, pp. 15-25.

[MRTL93] Marttiin, P.; M. Rossi; V.-P. Tahvanainen; K. Lyytinen: A Comparative Review of CASE Shells: A Preliminary Framework and Research Outcomes. In: Information & Management 25 (1993), pp. 11-31.

[SLTM91] Smolander, K.; K. Lyytinen; V.-P. Tahvanainen; P. Marttiin: MetaEdit – A Flexible Graphical Environment for Methodology Modelling. In: Advanced Information Systems Engineering, Proceedings of the Third International Conference CAiSE'91. R. Andersen, J. A. Bubenko jr. and A. Solvberg (eds.). Berlin, Springer-Verlag, (1991), pp. 168-193.

[SoTM88] Sorenson, P. G.; J.-P. Tremblay; A. J. McAllister: The Metaview System for Many Specification Environments. In: IEEE Software 14(1988) 3, pp. 30-38.

[Strö05] Ströbele, T.: EclipseUML – UML and Eclipse. In: OOP, Münich, 2005.

[TMHY80] Teichroew, D.; P. Macasovic; E. Hershey; Y. Yamamoto: Application of the Entity-Relationship Approach to Information Processing Systems Modeling. In: Entity-Relationship Approach to Systems Analysis and Design. P. P. Chen, North-Holland, 1980, pp. 15-38.

[VeSr95] Vessey, I.; A. P. Sravanapudi: CASE tools as collaborative support technologies. In: CACM 38(1995) 1: pp. 83-95.

[Wint02] Winter, A.: Exchanging Graphs with GXL. In: Proceedings of Graph Drawing – 9th Interational Symposium, Vienna, Springer Verlag, 2002.

[Zhan00] Zhang, Z.: Defining components in a MetaCASE environment. In: Proceedings of CAiSE'00, Stockholm, Sweden, Springer-Verlag, 2000.

Steven Kelly, Juha-Pekka Tolvanen

MetaCase
Ylistönmäentie 31
FI-40500 Jyväskylä
Finland
stevek@metacase.com, jpt@metacase.com
http://www.metacase.com

Matti Rossi

Helsinki School of Economics
Department of Business Technology
P.O. Box 1210
FI-00101 Helsinki, Finland
matti.rossi@hkkk.fi
http://www.hkkk.fi/~mrossi