JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2003; **15**:345–373 (DOI: 10.1002/smr.279)

Research

Dimensions of reengineering environment infrastructures

S. Ducasse[1] and S. Tichelaar[2,*,†]

[1] *Software Composition Group, Institute for Applied Mathematics and Computer Science, University of Berne, Neubrückstrasse 10, CH-3012 Berne, Switzerland*
[2] *Daedalos Consulting AG, Zürichbergstrasse 7, CH-8032 Zurich, Switzerland*

SUMMARY

Over the last decade many research groups and commercial companies have been developing reengineering environments. However, many design decisions such as support for multiple models, incremental loading of information, tool integration, entity grouping, and their impacts on the underlying meta-model and resulting environment have remained implicit. Based on the experience accumulated while developing the Moose reengineering environment and on a survey of reengineering environments, we present a design space defined by a set of criteria that makes explicit the different options and especially their dependencies and trade-offs. Using this design space, developers of future environments should have a better understanding of the problems they face and the impact of design choices. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: reengineering environments; reverse engineering; meta-modeling; code representation; tool integration; exchange formats

1. INTRODUCTION

The ability to reengineer legacy systems has become a vital matter in today's software industry. Software systems exceeding a certain critical size easily become difficult to maintain and adapt. Requirements change, platforms change and if a system does not evolve properly, its usefulness will decay over time [1]. Reengineering large industrial software systems is impossible without appropriate tool support. First of all, there is the scalability issue (millions of lines of code are the norm rather than the exception) and there is also the extra complexity of supporting and combining multiple tools with a wide variety of tasks (standard forward engineering techniques must be combined with additional reverse- and reengineering skills). The need for tool support in reengineering is reflected by the numerous tools or environments available in the reengineering research community [2–4].

*Correspondence to: S. Tichelaar, Daedalos Consulting AG, Zürichbergstrasse 7, CH-8032 Zurich, Switzerland.
†E-mail: sander.tichelaar@daedalos.com

Contract/grant sponsor: Swiss National Science Foundation; contract/grant number: Tools and Techniques for Decomposing and Composing Software (SNF 2000-067855.02) and Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1)

Received 8 January 2002
Revised 24 November 2002
Accepted 6 June 2003

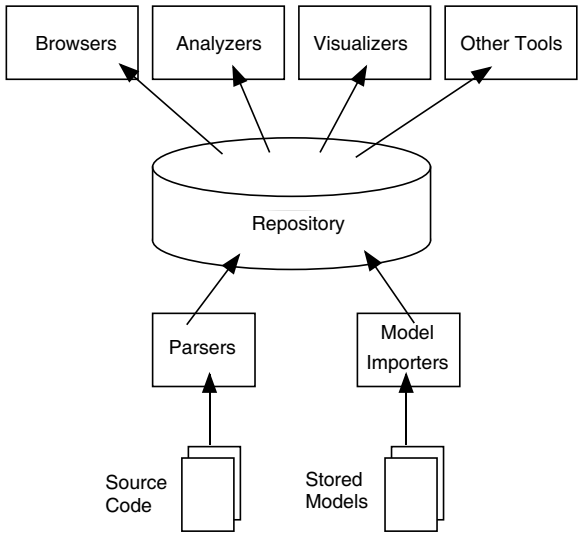

Figure 1. Standard structure of a reengineering environment.

Figure 1 shows the general structure of a reengineering environment. At the lowest level it shows the source code or stored models which can be imported into the reengineering environment by appropriated parsers. The middle level shows the repository, which contains an abstracted view of the source code. The top level shows the tools that use the repository as their information base. The repository's meta-model is the central part that lets everything work together. The properties of the repository, and thus of the complete environment, are highly influenced by the meta-model that describes what and in which way information is modeled. The meta-model not only determines if the right information is available to perform the intended reengineering tasks, but also influences issues such as scalability, extensibility, support for multiple models, and information exchange. We name *infra-structural* aspects the various aspects of the meta-model that have an impact on the environment functionalities.

The problem

Apart from a few exceptions, the documentation of existing meta-models and tool environments does not describe why the modeled information is relevant for a given set of reengineering activities [5–7]. Especially, no trade-off analysis exists of the different modeling alternatives that one may choose from. For example, the way entities are referencing each other may forbid the possibility to fuse different views of the same application created by different tools. Therefore developers of new reengineering environments do not have a conceptual tool to help them identify the problems encountered when building environments. They cannot easily assess the possible solutions and their tradeoffs.

The contribution

The goal of this article is to identify important infra-structural aspects of building reengineering environments with a focus on meta-model design. It presents a conceptual space in the sense of Lane or Wegner's work [8,9] defined by a set of criteria that identify the important questions that have to be answered when designing and building a reengineering environment. We identify the crucial design aspects that a reengineering environment developer has to face, the choices that are available for each aspect and the implementation solutions including their impact and interrelationships.

The design space presented is the result of the analysis of well-known reengineering environments such as PBS [10], Datrix [11,12] and Rigi [13,14] that we performed while building our own environment. It is also based on our five years of experience designing the FAMIX meta-model [15] and developing a number of tools [16–18] on top of the Moose reengineering environment [20].

It is out of the scope of this paper to come up with a design for a specific reengineering environment such as Datrix [11] or Rigi [13,14] and their underlying meta-models. Hence we do not cover the ability of an environment to support specific reengineering tasks and the exact contents of a specific meta-model. However the criteria we identify capture the variety of reengineering activities.

The rest of the paper starts with an introduction of the design space (Section 2). In the following sections we describe the criteria in detail (Sections 3–11). In Section 12 we validate the criteria by showing how four reengineering environments, namely Datrix [11], Rigi [14], SOUL [21–23], and Moose [17,20] fit into the space. Finally, in Sections 13 and 14, we survey related work and give our conclusions.

2. INTRODUCING THE DESIGN SPACE

First we clarify the concept of design space. In this article we follow the definition given by Lane [8]. In his work on system architectures, Lane refers to a design space as follows.

> '. . . a multi-dimensional design space that classifies system architectures. Each dimension of a design space describes a variation in one system characteristic or design choice. [. . .] Another way in which a design space differs from geometric intuition is that the dimensions may not be independent. In fact, it is important to discover the correlations between the dimensions in order to create design rules describing appropriate or inappropriate combinations of choices.'

To illustrate the kind of support we expect from a reengineering environment, we start with a short scenario. Next, we summarize the requirements for a reengineering environment in a list of questions that characterize the design space [8,9]. We are only focusing on the environment aspects that have an impact on the design of the underlying meta-model. Lethbridge *et al.* [5] have proposed a more complete list of requirements for a reengineering platform such as the response time of queries or support for frequently used facilities but they did not address the infrastructural aspects we present in this article. Finally, we show an overview of the design space. Each of the space axes is discussed in detail in the subsequent sections.

2.1. A scenario and its analysis

We paint here a typical usage scenario of a reengineering tool environment.

> The reengineer, let's call her Claudia, is confronted with a legacy system written in C++ of about one million lines of code. Her intention is to extract the architecture and discern possible problems in changing it. First she extracts information from the C++ system into a repository. Because she is interested in the high-level architecture she decides to only extract program entity information such as the structs, classes, methods and instance variables. She applies a tool to visualize the structure of the application and uses grouping techniques to collapse classes into a higher-level view.
>
> Claudia has a third-party metrics tool that can help her in understanding the system. Claudia exports the information from her environment into a standard exchange format and imports it into her metrics tool. She computes some metrics, exports them back to the environment where she combines the obtained results to enhance the views she gets with size information. She detects a big class in module *X* on which many classes in other modules depend. Looking at the source code she finds that some of the functionality of this class can be distributed over the different modules.

The scenario illustrates the following design aspects of a reengineering environment: the extraction level of detail (Claudia chooses the program entity level), different kinds of entities (she produces architectural entities), the use of grouping (she groups entities to produce a modular view), the tool integration, information exchange and incremental loading (she uses an integrated visualizer, an external metrics tool, an interchange format and merged metric results), and the annotation of entities (the metrics are associated with the entities they relate to).

Naturally, the scenario only covers a subset of reengineering activities. For instance, Claudia might want to apply similar analyses on other implementation languages, compare multiple versions of the same system, refactor the code etc. We summarize these and other issues in Section 2.2.

2.2. Requirement issues summarized

We summarize the issues involved when building a reengineering environment by introducing a list of questions that a developer of reengineering environments typically needs to answer.

2.2.1. Questions concerning the information to be modeled

Language support. How many implementation language(s) must be supported?

Level of detail. How detailed should the extracted information be (i.e., full parse-tree, main program entities, architecture)?

Model completeness. Can you work with incomplete models? Can the environment deal with only having partial representation of the code? How precise must your parser be?

Multiple models. Do you need to represent the software system at several levels of abstraction (code, design, analysis)? Will you analyze several releases of the same software system?

Scalability. How large are the programs[‡] you deal with? How many versions have been released? How many models do you need to extract?

2.2.2. Questions concerning the tasks to be performed

Grouping. Will you create higher level abstractions by grouping model elements? Do you need to group these groups? Do you need to group elements belonging to multiple models? Do you need to group relationships? Can you represent entities that do not directly represent source code entities? Will you exchange such entities and group them?

Tool integration. Must the tool environment exchange information with other tools? How do you merge information coming from different tools?

Extensibility. Must the meta-model be able to accommodate new kinds of information? Is it needed to annotate model elements? Must the environment adapt itself to new kinds of information?

2.2.3. Questions concerning infrastructure

In addition to the issues brought forward by the explicit requirements in Sections 2.2.1 and 2.2.2, some key infrastructure areas need attention, because implementation options can make or break the ability to fulfill a requirement. These infrastructure issues are as follows.

Exchange format. Does the format need to be easily human-readable? Does the file format support concatenation? How precisely should it reflect the internal data-structure? Must an industry standard be supported?

Entity reference. How are model elements identified (names, unique identifiers, universally unique identifiers)? Should the reference schema handle references over multiple models? Should groups of entities be referenced? Can it handle multiple models?

Meta-modeling. How do you represent and store information about a model (e.g., name of the creating tool, the level of detail of the extracted information)? Can a model easily be extended (e.g., with annotations, new attributes, new type of entities)?

2.3. Design space in a nutshell

The issues brought up in Section 2.2 do not stand alone. A choice made for one issue often influences the choices for another one. An example is the level of detail. When the tool-set supports the abstract syntax tree level of information rather than the program entity level, it is much harder to support

[‡]The word 'program' refers to any unit of source code from a complete application to a single file that the environment has to represent.

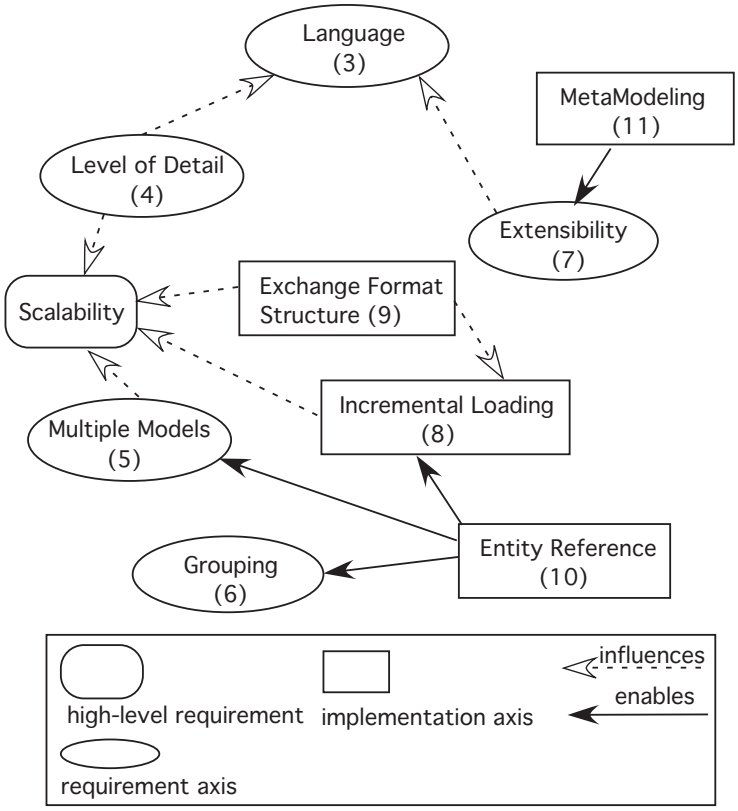

Figure 2. Overview of the design space. The numbers point to the sections that describe the axes.

multiple languages, because it is easier to map multiple languages to a more abstract higher-level representation than to a very detailed one.

We capture the correlations or dependencies between the choices in a *design space* [9,8]. Within the space one reengineering platform is determined by the values chosen for each axis. The space consists of *requirement axes* covering decisions that are dependent on the user requirements that the environment and its underlying meta-model must fulfill (presented by the questions of Sections 2.2.1 and 2.2.2), and *implementation axes* covering issues that are not directly reflected in the user requirements (presented by the questions of Section 2.2.3). Figure 2 shows the design space with the high-level requirements, the axes, and how they interrelate. This figure acts as a roadmap for the axes presentation.

The rest of this paper discusses the axes in detail. Every axis consists of an introduction that presents the main choices, the dependencies with other axes or requirements and implementation issues that discuss the impact of implementation decisions in more detail.

3. LANGUAGE AXIS

The choice of which language(s) to support is largely driven by the requirements of the end-user. However, from an environment developer point of view, the key issue is whether only one language or several languages should be represented. The real problem occurs when several languages use similar concepts with different semantics.

Dependencies

Level of detail. The greater the level of detail, the higher the reasoning power you get, but the harder it is to support multiple languages in a common way (see Section 4).

Extensibility. If the meta-model consists of a common core with language extensions, the appropriate extension mechanisms must be available (see Section 7).

Implementation issues

If only one language needs to be supported, the meta-model typically contains constructs of that language in a straightforward one-to-one mapping. If multiple languages are supported, the constructs of both languages are modeled either separately or using a common abstraction. Separate modeling is typical in the case of languages that have dissimilar paradigms. In such a case the meta-model is often constructed with separate, but connected sub meta-models for every paradigm [24]. If the supported languages have the same or an overlapping paradigm, common constructs are often modeled with a single abstraction [11,12]. The typical structure of such a meta-model is a language-independent core with multiple language extensions. For instance, a core could contain a Class abstraction which would allow class concepts in Java, Smalltalk and C++ uniformly, but the C++ template would be modeled in the C++ language extension. Treating similar constructs in a similar way results in language independence and reuse of analysis code. On the other hand, treating them explicitly decreases problems with semantic differences [25].

It is often useful to store the mappings. For example if Java interfaces and classes are modeled using a common Class abstraction, the information of whether the element represents a class or interface is stored. This allows, for instance, visualization tools to color Java interfaces differently from Java classes. Language-independent tools, however, can just treat the common concept without having to know about the language details. For similar reasons, if the modeled system is implemented in multiple implementation languages, it is often necessary to record the implementation language for every entity or every group of entities.

4. LEVEL OF DETAIL AXIS

'Level of detail' represents with which precision the source code is modeled. It is well accepted to distinguish the following levels of detail:

1. Abstract syntax tree (AST) level information—a complete view of the source code. It is normally detailed enough to regenerate the source code and sufficient for control-flow analysis.

Table I. File sizes for different levels of detail of Swing 1.3.0.

Swing 1.3.0: 225 KLOC 2700 classes	CDIF file, MB (zipped MB)	XMI file, MB (zipped MB)
Classes, methods, attributes	8.3 (0.53)	21.8 (0.90)
+invocations and accesses	12.1 (0.82)	28.8 (1.25)
+formal parameters	15.5 (0.99)	39.7 (1.67)
+metric information		84.7 (3.15)

2. Program entity level information—abstracted but factual view on source code: classes, methods, functions, etc. It is normally used for generating structural views on the target system. Typically only limited information about method bodies is available, consisting of method invocation and variable access information, which is sufficient for dependency analysis.
3. Architectural information—modules, packages, elements grouped in directories, COM or CORBA components, architectural and high level views on the program entity level.

The analysis that the environment should support directly influences the level of detail of the source code representation. For example program slicing and data flow analysis [26,27] require a greater level of detail than the program visualization used in CodeCrawler [16] or Rigi [13,14].

Dependencies

Scalability. The greater the level of detail, the higher the memory consumption and load time of information from databases or files and the slower the response times of tools that use the information. To give an idea for the impact of different detail levels on resource consumption, we show the file sizes of the textual representation of a model using our own FAMIX meta-model (see Section 12.1) using two standard exchange formats, namely CDIF and XMI. The numbers are shown in Table I. The default representation of both standards is not optimized for space consumption, which is confirmed by the high level of compression achieved by zipping the files. The modeled system is the Java Swing framework, version 1.3.0, consisting of 7.2 MB of source code (225 KLOC). It has been parsed by the SNiFF+ [28]) parser, version 3.2.1. It consists of ∼2700 classes (including inner classes) and ∼11 500 methods. The metric information contains up to 25 metrics per source code entity where every measurement is stored as a separate element in the file.

Implementation issues

There are several techniques for dealing with large amounts of information: The first is *collapsing information* to reduce memory footprint. For instance, invocations of methods in one class to methods in another class can be modeled one by one, or collapsed into a single invocation relation between the two classes. The second one is to use lazy representation/loading. For example, *incremental extraction* can be implemented by using source anchors, i.e., pointers to the original source code, that allow one

to go back to the source code and extract any additional information when needed. It is important that the partial information is still sound. For instance, if only method information is stored, information about the containing classes should typically be stored as well.

Two important points should be considered when using source anchors. First, using source anchors may require the use of incremental parsers. Second, the granularity of the referenced source code segment should not be too small to avoid difficulties while extracting the information. For example, in the analysis of object-oriented code, source anchors associated with methods offer a good granularity. In such a case method invocations and attribute accesses can be represented, while complete method bodies are not directly represented, but accessible via the source anchors.

5. MULTIPLE MODEL AXIS

The possibility of analyzing multiple models simultaneously is useful in evolution analysis, where multiple versions of the same system need to be analyzed [29,30]. Likewise, it is interesting to analyze parallel branches of similar applications, for instance, to develop a framework by abstracting common assets in these branches. Another aspect is that different models can have different meta-models. This is typically the case when multiple paradigms are modeled [24].

Dependencies

Entity reference. Multiple models require the possibility to uniquely identify elements from different models, even if these elements are similar, for instance, the same class in multiple versions of the same system. Furthermore, it must be possible to identify which model an element belongs to (see Section 10).

Meta-modeling. Information about models themselves (represented system, meta-model version...) can be modeled as part of the model itself or as a meta-entity (see Section 11).

Implementation issues

Instead of making multiple models explicitly part of the meta-model, a tool can support multiple models in its implementation. This keeps the meta-model simpler, but makes exchanging entities of several models between tools *ad hoc*. Tools are dependent on (possibly multiple different) implementations rather than one underlying meta-model.

6. GROUPING AXIS

Grouping several entities together is an important technique in reengineering, primarily to build higher-level abstractions from low-level program elements or to categorize elements with a certain property [13,14]. Two ways of grouping can be identified: intentional (description-based) and extensional grouping where the group acts as a bag of (references to) model elements [31,22].

The following considerations should be evaluated that have an impact on entity reference (see Section 10):

- nested groups, i.e., groups that contain other groups.
- which entities can be grouped, e.g., can a group only contain named entities such as classes or also nameless entities such as relationships.
- groups of entities over multiple models, i.e., can a group contain entities that belong to different models.

Dependencies

Entity reference. The entities that need to be groupable must be uniquely referable. This can include the groups themselves, similar entities in multiple models and associations between entities (see Section 10).

Implementation issues. If intentional groups are to be supported, the model must define how descriptions are modeled. OCL expressions [32] or other formalisms could be used. Furthermore, intentional groups need strategies for recomputing the contents when a model is adapted or additional information is incrementally loaded [22]. This all is, however, a separate topic that goes beyond the scope of this paper.

7. EXTENSIBILITY AXIS

Extensibility is an important issue in modeling software as it allows additions to a model without having to change the model itself. Typically extensibility is needed for the following kinds of information:

- *Language-specific information.* In the case in which a meta-model is used with a language-independent core, it is often still interesting to store language-specific information. For instance, a visualization tool might color all Java interfaces, even if they are mapped to a common class concept in the language-independent core. In addition, typically languages have their own specific problems that are interesting in themselves. An example is the analysis of include hierarchies in C++.
- *Tool-specific information.* Tools might store and exchange tool-specific information such as analysis results or layout information for graphs.
- *Whatever information people find worth modeling.* Not all information needs can be known in advance. Furthermore, light-weight approaches for annotating model elements help record the information gained in the iterative process of understanding a system.

Dependencies

Meta-meta-modeling. There is a dependency between this axis and the meta-meta-model axis as the mechanisms described in the following could be specified by the meta-meta-models of reengineering environments. In particular the extensibility of the meta-model is restricted by the extensibility the meta-meta-model allows one to have. This is especially an issue when using modeling standards such

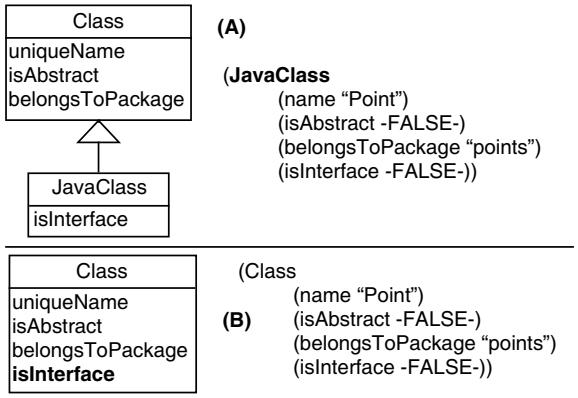

Figure 3. Extending Class with an attribute isInterface by: (a) subclassing a Class; and (b) adding the attribute to the existing Class element.

as the MOF [33], because they restrict you to their extension mechanisms. The MOF and GXL [34] do not support class extensions, i.e., the ability to add attributes to existing classes. CDIF [35] and RDF [36] both allow this kind of extension.

The following subsections describe the extension mechanisms to be considered: adding new entities to a model, adding attributes to existing entities, and annotating entities.

7.1. Adding new entities to a model

Being able to add new entities representing constructs not already represented and that cannot be mapped in a sensible manner to existing entities is a major extensibility capability of the model. This possibility is important when modeling multiple languages with a common language-independent core.

7.2. Adding attributes to existing entities

For certain purposes it is required to add attributes to existing entities. An example is the information that a class in a model represents a Java interface. The Java extension adds an attribute isInterface to the Class entity. This can be done through subclassing or by explicitly adding an attribute. Figure 3 shows an example of this. The subclassing solution: (a) demands for all clients to know the extension and how it relates to the original entity. It demands importers that are fully meta-model aware and can deal with extensions that it does not know yet. When the attribute is just added to the original entity (b) an importer that does not (want to) know the extension just recognizes the element it knows and can ignore any attribute of the element it does not recognize. A more severe problem with the subclassing solution is the fact that multiple orthogonal extensions (typically language and tool extensions) might exist which cannot be modeled by inheritance at all. This is illustrated in Figure 4. This is also a problem

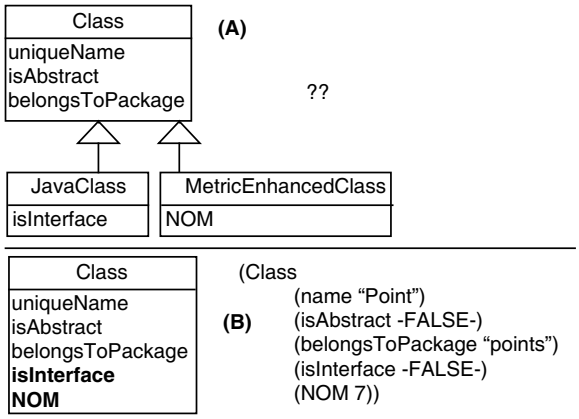

Figure 4. Extending Class with two orthogonal attributes by: (a) subclassing a Class; and (b) adding the attributes to the existing Class element.

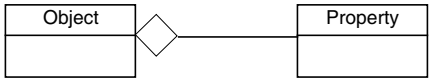

Figure 5. Simple extensibility: object annotation.

if an attribute must be added in an existing hierarchy where all existing subclasses should inherit the attribute.

7.3. Annotating entities

Annotations allow any information to be added to any entity. Entity properties, analysis results, or notes can then be attached to the entity they refer to. Although a meta-meta-model could define a way to support annotations, it is often considered the domain of the meta-model. In such a case the meta-model explicitly allows any entity to be in relation with some property, annotation or tag objects (see Figure 5).

Implementation issues

The questions of the representation of the annotation object in memory and in the exchange format have to be answered. From our experience of representing metrics as entity annotations, we learned that annotation objects can constitute more than 90% of the data processed while loading and analyzing

a system. One solution is to only represent properties explicitly in the exchange format and use appropriately optimized data structures for the in-memory representation.

In the exchange format, annotations can be represented as information of an annotated entity or as an explicit entity. The first solution precludes loading of annotations separate from their containers, but requires less space as illustrated below:

Annotation in the entity itself

```
(Class
  (name A)
  (annotation NumberOfMethods 45))
```

Annotation associated with the entity

```
(Class
  (name A))
(Annotation
  (belongsTo A)
  (NumberOfMethods 45))
```

8. INCREMENTAL LOADING AXIS

Incremental loading of information is about the ability to load new entities or additional information for entities that already exist in a model. The reasons for considering incremental loading are resource optimization and the merging of information from different sources. It generally allows the model to contain references to information that is not in the current model and to load only parts of models. It also allows extractors to extract partial information and the integration of information generated by different tools.

Dependencies

Entity reference. The reference schema must allow information elements to be able to reference each other in different files (see Section 10).

Scalability. Incremental loading can be used for resource optimization. Information about the same system can be stored (and thus transferred) in separate files or databases. Furthermore, information can be loaded on demand. For example, computed metrics or analysis results can be so space-consuming that you only want to load them when necessary.

Implementation issues

Referenced information which is defined in a separate resource must be able to be filled in later and be correctly associated with the entity it refers to. A solution is to use stubs, i.e., empty placeholders, that represent the missing entities. If later the actual element is loaded, it replaces its stub. Note that referenced information often is deliberately left absent. Frequently we want to model an application but not the complete libraries or frameworks it uses. In such cases the model ends up with references to framework entities that are not represented in the model.

9. EXCHANGE FORMAT STRUCTURE AXIS

Repository information is often stored in text files. It is a lightweight way of storage, which is particularly well-suited for information exchange. Important criteria of the format which is used to store the information are human readability, lightweight manipulation, the ability to incrementally load information, and the non-influence of the entity loading order [12]. Saint-Denis *et al.* [37] describe a set of 13 criteria that are important for an exchange format. Many of the criteria they describe are non-functional, such as reliability and completeness, which rather depend on the quality of the tool that produces the information than an intrinsic property of the exchange format.

We discuss here only three aspects that depend on the actual structure of the exchange format because the structure has an impact on the functionality of the environment such as supporting multi-models or incremental loading. Elements can be stored in a *nested* way, saved in, what we call, *chunks* or in a completely *flat* way. This is independent of the quality of the extractor and the actual encoding (i.e., if the line ends with a bracket or an XML-tag). In the following examples we use ( ) to delimit stored elements.

Dependencies

Incremental loading. As explained later in this section the structure of the exchange format can prevent the incremental loading of models.

Implementation issues

Apart from the formats themselves in Section 9.1, we discuss the storage of multi-valued attributes in Section 9.2.

9.1. Nested, chunk, and flat formats

Nested. In the nested format every element physically contains its constituents. Relationships other than containment are modeled by explicit relationship elements. The following example shows a class A which inherits from class B and contains two methods (M and N) and an attribute (X).

```
(class A
  (method M
    (isAbstract true))
  (method N
    (isAbstract false))
  (attribute X
    (visibility public))
(inheritance
  (subclass A)
  (visibility protected)
  (precedence 2)
  (superclass B))
```

Evaluation. The nested format is complex and elements can quickly get large due to the elements they contain. It only favors human readability as long as the elements do not get too large. Incremental

loading is not well supported either, because elements cannot be stored without their containing element, and replacing an element requires finding it inside its containment hierarchy. The format is useful for a one-to-one representation of the tool-internal data-structure, which is typically optimized for information navigation rather than storage or exchange.

Chunk. In the chunk format, entities are not nested in their scoping entity. Simple relationships are stored as attributes of the entity an entity is linked to. In the example hereafter, the method M belongs to class A: both the method and the class are stored as explicit entities and the containment relationship is represented by the belongsToClass attribute of M. For relationships that need additional information to be stored, explicit entities are created. In the example, the inheritance relationship is an explicit entity with visibility and precedence as attributes.

```
(class A
  (isAbstract true))
(method M
  (belongsToClass A)
  (isAbstract true))
(attribute X
  (belongsToClass A)
  (visibility public))
(inheritance
  (subclass A)
  (visibility protected)
  (precedence 2)
  (superclass B))
```

Evaluation. The chunk format favors human readability, because entities are self-contained with the containment relationships readily available as attributes. The format supports incremental loading better than the nested format, because, when loading or updating an element, the containing entity does not need to be found or even be available. Because the format is less complex, importers can be simpler as well. The chunk format has as the main disadvantage that relationships are stored in two different ways.

Flat. The flat format explicitly represents all entities and relations, typically using a line-based format. For example, RSF or TA are based on this approach [14,38]. Contrary to the two other formats, which support entities with attributes in pair-value form, here the attributes require another storage form. In the pure form they are all explicit relations. Some formats are more elaborate and allow attributes as pairs on the same line [38]. This actually goes a step towards the chunk format. The following code shows the flat format with both kinds of attribute storage:

```
(class A)
(method M)
(methodBelongsToClass M A)
(attribute X)
(attributeBelongsToClass X A)
(inheritance A B)
(inheritance A B visibility protected)
(inheritance A B precedence 2)
```

Table II. Textual formats compared.

	Readability	Processability	Incremental loading
Nested	−	−	−
Chunk	+	+	+
Flat	−	++	+

Evaluation. With this approach, incremental loading is well supported and merging of models is simple. Machine processability is also simple, because the format is conceptually simple. Because it is line based, manipulation of models can be easily done by running a script (in Perl or similar scripting languages) to convert one file into another one. A drawback is that information representing one single entity can be scattered over the file, which hampers readability.

9.2. *N*-ary attributes

Supporting multi-valued attributes is necessary because some attributes intrinsically have multiple values. For example, a class may contain multiple comments and a method invocation typically has multiple polymorphically equivalent targets.

Each of the above formats can support multi-valued attributes. They can be represented either by a special kind of attribute that allows multiple values, or by multiple instances of a single attribute. For example, a class with multiple comments can be represented as follows in the nested and chunk format:

```
(class C
  (comments ``a comment'' ``another comment'')
  (attribute X))
```

Or similarly in the flat format:

```
(comments C ``a comment'' ``another comment'')
```

The second alternative, with multiple instances of a single attribute, again in the nested and chunk format:

```
(class C
  ...
  (comment ``a comment'')
  (comment ``another comment'')
  (attribute X))
```

Or similarly in the flat format:

```
(comment C ``a comment'')
(comment C ``another comment'')
```

Note that in all cases an importer needs to either be aware if an attribute can be multi-valued, or regard any attribute as potentially multi-valued.

9.3. Discussion

Table II summarizes the evaluation of the three format structures. The nested structure does not score well against our criteria. We do not recommend it for information exchange purposes, although it can be a viable option for complex data-structures. The chunk format does well in all three areas. The flat format is less readable, but simpler to process. Note that human readability gets less important when importing and saving tools stabilize. If that is the case it is less important for humans to follow what is going on.

10. ENTITY REFERENCE AXIS

The entity reference axis directly and indirectly influences many other axes in the design space. The decision how entities are referenced comes down to a trade-off between heavyweight solutions that ensure that every element in the model can be uniquely identified, always and everywhere between different tools, or solutions that are more light-weight, but might not adequately support grouping, multiple models, or referenced information that is not in the same file or database.

Dependencies

Grouping. The entities that need to be groupable must be uniquely identifiable. This can include the groups themselves, entities in multiple models and relationships (see Section 6).

Multiple models. Multiple models require the possibility to uniquely identify artifacts from different models, even if these artifacts are similar, for instance the same class in multiple versions of the same system (see Section 5).

Incremental loading. Incremental loading requires information elements in different files or databases to be able to reference each other (see Section 8).

Implementation issues

There are several possibilities to identify entities. One is to use the intrinsic unique name of entities that have one (like a class name or attribute name), or to use a meaningless unique identifier. We discuss both solutions in the next subsections and finish with a discussion.

10.1. Unique identifiers

One approach for model element identification is the use of meaningless identifiers[§]. Every model element gets such an identifier attached to it and all references use this identifier as well. The most important issue is how 'unique' this identifier needs to be: only within one database or file, only within a known set of files and databases, or always guaranteed unique. Light-weight approaches can be used,

[§]By meaningless, we mean identifiers whose format does not have semantics.

Table III. Textual representation of Swing 1.3.0 without and with UUIDs.

Swing 1.3.0	XMI file MB (zipped MB)	XMI + UUID MB (zipped MB)
Classes, methods, attributes	21.8 (0.90)	23.6 (0.93) (+8.3%)
+invocations and accesses	28.8 (1.25)	31.2 (1.30) (+8.3%)
+formal parameters	39.7 (1.67)	43.1 (1.75) (+8.6%)

the simplest one probably the use of integers. However, if uniqueness with a broader scope is required, solutions such as Universal Unique Identifiers (UUIDs) must be considered.

UUIDs are unique across both space and time, with respect to the space of all UUIDs. A UUID can be used for multiple purposes, from tagging objects with an extremely short lifetime, to reliably identifying very persistent objects across a network. UUIDs are generated using a combination of the network address and current time at the moment and place it was generated. Assuming that network addresses are unique and that time never runs backwards, this guarantees that UUIDs really are unique. Furthermore, the UUID generator must have access to an IP network address. An example of a UUID: `c842bf06-d202-0000-0282-5c410d000000`. However, using UUIDs increases memory consumption and loading and saving times. Table III shows, for the same system as discussed in Table I, the file sizes when using UUIDs and plain integers in XMI files. We see that the file size increases up to 9% when using UUIDs. Similarly the working memory usage of the same models in our reengineering environment (Moose, see Section 12.1) increased by up to 13%. For Smalltalk systems we measured memory usage increases of up to 40%. This is probably due to the fact that our Smalltalk parser extracts a lot more detailed information (such as arguments) than the parser we use for Java.

The complexity and resources needed to compute and store these identifiers might however be an overkill. Simpler approaches can be designed, but will rely on proprietary conventions.

10.2. Unique naming scheme

Another way of referencing elements is to reference them by their intrinsic unique names. For instance, the unique name of the class Box in the Java Swing library would have the unique name javax.swing.Box, which is known to be unique within a Java system, but not over time or different versions. Using unique names as identifiers has the advantage that reference information is humanly readable. A second advantage is that every time a model is generated from a certain system, exactly the same unique name will be used instead of generating a completely new identifier or having to take earlier assigned identifiers into account. A problem is that model elements that do not have an intrinsic unique name such as relations, models and groups, do not easily fit this scheme:

Relations. Relations can only partly be uniquely identified using names. A simple example is when class A inherits from class B, a unique name for the relationship could be A.InheritsFrom.B. However, this only works for one-to-one relationships and not for any relationship with multiple targets.

For instance, the invocation relationships resulting from a method that invokes another method twice, would result in two elements with the same name.

Multiple models. A unique naming scheme in the context of multiple models needs to take the model into account, especially if the different models model the same system (for instance, multiple versions of this system). In such a case the model should somehow be integrated in the unique name of an element. We could, for instance, append a model name to the intrinsic unique name. For example, consider the Box class in the Java 2 Swing library in version 1.3.0: Java2v130.javax.swing.Box. Note that uniqueness of model names depends on convention rather than on a language-induced rule.

Grouping. A problem similar to model names exists for groups. Although a group typically has a name to describe what is grouped, the uniqueness of those names is not intrinsic and therefore cannot be guaranteed. It is important to normalize unique names. The unique naming scheme should be clearly defined so that unique names are always the same. An example rule is that unnecessary spaces are always removed. If the meta-model supports different languages, unique names can either be close to every supported language, or a common scheme can be defined for all supported languages. The latter requires more thought about the naming scheme, but allows tools that are independent of the supported languages to deal with one naming scheme only.

10.3. Analysis

Unique names and meaningless identifiers have clear properties. Names can be used to reference entities with an intrinsic unique name, meaningless identifiers can be used to reference anything. The following example, expressed in simplified CDIF, shows the class Point in the HasMethod relationship.

```
(class
  (name Point))
(method
  (name intersect))
(Class.HasMethod.Entity Point intersect)
```

The same example using meaningless identifiers:

```
(class 111
  (name Point))
(method 112
  (name intersect))
(Class.HasMethod.Entity 111 112)
```

Another important difference is that unique names are reproducible and other identifiers not. Different tools generate different identifiers, which hampers incremental loading, because two model entities from different sources that represent the same code element cannot be linked through the unique identifiers. An interim solution can be chosen to leverage the best of both worlds: human-readable unique names for the named entities and some kind of unique id for the non-named entities. Such a combination preserves the advantages of naming and is able to uniquely identify any non-named element as well. Clearly, complexity increases when the two separate schemes are mixed.

A last solution is to provide both solutions and use them where appropriate. However, when an entity can be referenced in two different ways, incremental loading is not possible. If a certain entity is not present in the current model, it cannot be determined that a reference by name and a reference by unique identifier are actually supposed to reference the same entity.

Comparison with industry standards

CDIF [35] and XMI [39] promote an entity referencing schema that is internal to the file. As shown by the code example in Section 10.3, CDIF relationships use an identifier that is associated to the entity in the context of only one single file. However, if entities need to be able to be referenced over multiple files, the referencing scheme needs to be unique over those files, if incremental loading is to be supported. One solution is to use UUIDs. Yet another approach is taken by XLink [40], a standard for linking between elements in different XML files. It links elements in different resources by explicitly mentioning the resource the element is defined in. However, the exact location needs to be known, rather than a reference that can be in any (and multiple) files.

11. META-META-MODELING AXIS

Meta-meta-modeling is about making the representation of the meta-model explicit. Hence, a meta-meta-model allows one to reason about, and possibly change the meta-model. Properties of a meta-model, e.g., how it can be extended, can be defined using such an explicit meta-layer. The following options need consideration.

An explicit meta-meta-model

An explicit meta-meta-model results in a standard way of describing meta-models. This description can be used for the following capabilities:

- identify meta-models, i.e., are different tools compliant to the same or compatible meta-models. This issue can be solved simply by providing meta-information representing the meta-model.
- auto-adapt environments, i.e., tools can use an explicit meta-model description to customize themselves. For example, a user interface can show, or a saver can store, any information according to any meta-model, because it knows the structure of the meta-model through the meta-meta-model.
- create meta-models, i.e., meta-tools use the meta-model description to create meta-models themselves. This requires an approach where the meta-model description is interpreted to create appropriate meta-model representation. For instance, it is possible to interpret UML description based in its MOF [33] specification to create UML compliant tools.

A standard meta-meta-model

A standard meta-meta-model such as the MOF has the advantage that you do not have to design it yourself. Furthermore, it allows you to use compliant tools and other meta-models that adhere to

the standard. However, when you design your own meta-meta-model you have full control over its capabilities, for instance, that it supports all the extensibility mechanisms you need.

Dependencies

Extensibility. The extensibility capabilities of a meta-model are determined by its meta-meta-model. A standard meta-meta-model might restrict you too much (see Section 7).

Implementation issues

The implementation issues include the storage of model information and the use of explicit meta-model descriptions.

Model information. Knowing information such as the language, the dialect, the tool that extracted the information, the date of creation of the model, and the identification of the meta-model to which the model conforms, are all important information to precisely identify a model. Such model information can be represented in two different ways:

- in the meta-model. For example, headers of CDIF exchange files include such kinds of information in addition to the complete description of the meta-model itself described in terms of the CDIF meta-meta-model [35].
- as a common entity of the model. This approach is simpler since the model information is treated as any model element. It is also independent of any meta-meta-model.

Meta-model creation and tool adaptation

Using a meta-meta-model to generate models is not complex as long as it deals with the structural aspects of the meta-model such as navigation among entities. However, the specification of entity behavior, e.g., how to compute all inherited methods of a certain class, requires a language that is able to formally express such behavior in terms of the meta-model.

12. VALIDATION

The goal of this article is that the criteria it presents helps the designer and implementors of future reengineering environments. Clearly such a claim cannot be scientifically validated because we do not have access to internal design decisions of reengineering environments. We validate the design space in another, less satisfying, way. We have looked at the existing literature and especially extracted as much information as possible about three specific reengineering environments. Together with an evaluation of our own reengineering environment, we have used this information to determine if the criteria described in this paper can effectively characterize reengineering environments.

However, extracting information from the literature is difficult, because there is always the possibility of missing aspects or being inexact. Furthermore, the literature contains a lot of work related to specific issues such as exchange formats [12, 34, 37, 38, 42] and higher-level comparisons of tools at the level of the tool functionality [2–4]. There is nearly no description and analysis of the trade-offs made while

developing the meta-model and the reengineering environments. Only [5, 7, 11] elaborate on certain aspects of the modeled information but in a rather limited way because the criteria we describe are often related to the internal aspects of reengineering environments.

The rest of this section discusses the tools in detail and evaluates the design space in the context of these tools.

12.1. Datrix, Rigi, SOUL, and Moose

We have chosen four existing tools with different flavors to have a broad scope for the validation. Datrix is a reengineering environment with complete source code representation that supports the analysis of C, C++, and Java. Rigi is a reverse engineering tool manipulating graphs that represent information extracted from source code. SOUL is a reflective environment based on logic programming that allows the analysis of Smalltalk applications. Table IV shows an overview of these tools in the context of the design space.

Datrix. Datrix[¶] is a source code analysis tool that has a complete source code representation in the form of enriched ASTs and supports code based analysis [11,12]. Moreover, the Datrix meta-model was designed with the intention that it could serve different languages. Datrix uses the TA flat format [38] as an exchange format. It is worth mentioning that Datrix fits well into the criteria we presented. Lapierre *et al.*'s [12] is one of the few articles that discusses issues learned from building a reengineering environment. It raises the issue of entity reference naming even if it does not take a clear position. Datrix supports AST manipulation but does not define refactorings.

Rigi. Rigi is a well-known *reverse engineering* tool [13,14]. It is interesting for our evaluation because it does not aim to represent complete and concrete source code but rather the manipulation of views on graphs that represent part of the code or any other information. Usually users extract information about the code and represent it in the Rigi Standard Format, a *flat* format. This information is then loaded and manipulated in Rigi. Rigi does not include any specific information about the language or application analyzed, even if some domain information description can be provided. Contrary to Datrix or Moose, Rigi does not have entities representing languages constructs but manipulates entity–relation graphs. Rigi supports grouping as an important means of manipulating graphs.

SOUL. SOUL (Smalltalk Open Unification Language) is a logic programming language living in symbiosis with its implementation language (Smalltalk) [21–23]. The integration between the two languages from different paradigms allows one to write logic programs that can do full logic reasoning on objects and do full logic reasoning using objects. This was primarily exploited to reason on static information (since in Smalltalk classes and methods are first-class objects), but also to reason about dynamic information gathered after instrumenting a system. Static code reasoning is performed using the LiCoR framework (Library for Code Reasoning) that is built on top of SOUL and represents Smalltalk applications through a complete representation of the parse tree.

SOUL currently supports only Smalltalk. Since LiCoR is meant to reason about code in the current Smalltalk image, it does not support different models. However, it features all the reasoning and

[¶]The Datrix project at Bell Labs has ended, so some internal documents are no longer available.

Table IV. Four reengineering tools evaluated.

Tool/design space axis	Datrix	Rigi	SOUL	Moose
Languages	C, C++, Java	Any	Smalltalk	C++, Java, Smalltalk, Cobol
Level of detail	Program level entity. Complete source code as annotated ASTs	Various. Extracted information should be mapped to nodes and edges	Program level entity. Complete source code as annotated ASTs	Program level entity. independent OO core. No method body but local variable, attribute accesses, and method invocation
Multiple models	No	No	No	Multi-model supported at the environment, not by meta-model
Grouping	Not found	Major feature. Not between different models	By extension and intension	Grouping over multiple models
Extensibility	Represent similar languages. Any entity can be annotated	Represent similar languages	Any entity can be annotated	Extensible core. Any entity can be annotated
Incremental loading	Not found	Yes	No	Yes
Exchange format structure	Flat format (TA) [42]	Flat format (RSF)	Not intended to be exchanged with other tools. Uses Smalltalk code file out format	Chunk format (CDIF and XMI based)
Entity reference	[12] mentions that this issue is under discussion	By naming in the explicit meta-models	Using logic variables or object pointers	UUIDs and unique-Name
Meta-meta-model/model information	Not explicit/no model information	ER/No model information	Both explicitly represented by logic facts	ER used for meta model generation and generic tools/ model information as a model entity

abstraction power of a logic programming language to do grouping. Hence it has support for intentional and extensional grouping, which is used to express design patterns and software architectures. Because of its integration with Smalltalk, the model is extensible. When the model needs to encompass new kinds of entities, only some reification rules have to be added to make those entities explicit in SOUL. It supports incremental loading by definition: during the logic inference process those parts of the system that are needed are reified on the fly. Extra model information can be added easily in the form of extra logic facts and rules. Referencing of elements in the model is done as in any other logic programming language, i.e., by the use of logic variables that reference Smalltalk entities. Finally, SOUL supports code refactoring.

FAMIX and Moose. FAMIX is the meta-model we designed for modeling software [15] and the Moose Reengineering Environment is the platform we built with a repository based on this meta-model [20]. Moose supports reengineering of medium to large-sized industrial software systems in multiple object-oriented implementation languages. Its meta-model, FAMIX, defines a minimal language-independent core of information that is always available and on top of which additional levels of information can be added [43–46]. The supported languages are modeled at the *program entity* level. The meta-model defines a minimal core based on object-oriented concepts and can be extended to support new entities found in different languages or needed when supporting higher views of the applications.

Model elements can be either referenced by unique name for the elements that have one (the entities), or by unique identifier which all elements have (both entities and relations). FAMIX illustrates the strong impact of choices for the referencing schema. The unique referencing schema supports grouping even over multiple models. The meta-model does not support grouping explicitly, but the environment requires that all entities have a unique name. Moose supports incremental loading. It uses standard exchange formats such as CDIF and XMI, but with the chunk referencing schema. Apart from the textual storage with CDIF and XMI, Moose can store its in-memory repository. The FAMIX meta-model does not support multiple models, but Moose does. The repository can hold multiple models, but textual information transfers (defined by the meta-model) do not contain information about multiple models.

Moose defines its own simple ER-like meta-meta model. This meta-meta model is used to generate meta-models and support generic tool facilities. FAMIX originally represented model information as part of its meta-model. Now we treat the meta-model information as an entity, because it avoids having a meta-model-specific section while loading models. However, the model and information about the model are mixed, which is conceptually a questionable design choice. Hence tools have to be aware that one entity representing the model information exists. Furthermore, entities can be annotated.

Finally Moose provides experimental support for higher level refactorings based on program entity level information rather than the more complete AST level of information [25].

12.2. Evaluation

The four evaluated reengineering environments cover a broad spectrum. The design space characterizes them from the more general (Rigi with its general meta-meta-model, potentially supporting any language) to the more specific (SOUL with its high-detail modeling of Smalltalk in logic facts at the AST level). Furthermore, the paper defines, as can be seen in Figure 2, three dependency hotspots,

namely scalability, entity reference, and extensibility. The information from Table IV illustrates these hotspots with examples.

Scalability. The scalability generally depends on the amount of information that is modeled. This is influenced by:

- the level of detail. With Rigi this issue is dependent on the actual meta-model used. Moose uses the program entity level, and scales to systems over a million lines of code. SOUL and Datrix model systems are at the AST level. Inherent to the higher level of detail these tools scale less easily, SOUL was used to analyze industrial frameworks of a couple of hundred classes. There are no numbers for Datrix. The only information we have is that it was used to analyze industrial applications.
- multiple models. The more models, the more information must be dealt with. Only Moose supports multiple models, mainly for evolution analysis [29].
- incremental loading, which allows us to focus on the information in the current interest.

Reference schema. The reference schema influences the ability of a tool to group entities, incrementally load information, and support multiple models. The examples in Table IV confirm this. A Soul model references the original code elements directly. There is no defined unique naming scheme and thus Soul does not support exchanging and incremental loading of information. Moose and several other tools based on FAMIX use a strict unique scheme, allowing for the incremental combination of different sources of information for the same software system. Also information about groups of elements can be explicitly defined without relying on the specific data representation of a tool. Rigi delegates this issue to providers of specific meta-models. Datrix also does not provide any strict definition, although the issue has been under discussion [12].

Extensibility. Meta-meta-modeling influences the extensibility by explicitly defining an extension interface. Rigi only provides a meta-meta-model and is inherently extensible. Moose has an explicit metamodel as well. It supports extension of existing and extension by new elements as well as the ability to annotate any element. Soul has a meta-meta-model based on logic facts. Datrix's model is not explicitly extensible. It provides the ability to annotate elements. Second, extensibility influences how well differences between languages and paradigms can be modelled. Well-defined extensibility also supports the integration of tools, because commonalities and differences can be explicitly dealt with.

The data show that indeed the space captures the inherent property dependencies. Furthermore it tells us certain properties imply certain (im)possibilities. For instance, SOUL is language specific which implies low interoperability. Therefore we can conclude that the space, i.e., the categorization together with dependencies, can provide us with a better understanding and additional insights into the presented reengineering environments.

13. RELATED WORK

There exist a number of models that support software representation, object-oriented or not. One kind of models are those aimed at object-oriented analysis and design (OOAD), the most notable example being the Unified Modeling Language (UML) [47]. However, these models represent software at the

model or design level. Reengineering requires information about software at the source code level. The starting point is the software itself demanding a precise mapping of the software to a model rather than a design model that might have been implemented in many different ways [48]. In the reengineering research community several models exist that model the code itself. They are aimed at procedural languages (Bauhaus [41]), object-oriented/procedural hybrid languages (TA++ [49], Datrix [6,11], FAMIX [48]) and multi-paradigm models [24]. Most models support multiple languages, either implicitly or explicitly.

Little analysis about the design and implementation of reengineering environments exists [5–7]. Koschke's thesis discusses the relevance for modeled information of the Bauhaus Resource Graph model for reengineering C programs [7]. Likewise, Datrix has a clear description of what is modeled, namely abstract syntax trees with added semantic information [6]. Basically complete programs are modeled in all possible detail. The relevance question is not discussed, as the goal is to model everything and make the availability of the original source code irrelevant for the analysis tasks. Lethbridge and Anquetil present a list of requirements for reengineering platforms and present the design of the meta-model they developed [5].

Saint-Denis *et al.* provide a list of evaluation criteria for model interchange formats, including an analysis of how various inter-exchange formats evaluate against those criteria [37]. Many similar issues arise, such as scalability, simplicity and readability. Our work is different, firstly, because the textual exchange format is only one of many things discussed in this paper, secondly in that we discuss in much more detail practical issues such as reference schemas, unique identification and meta-model extensibility. The Graph eXchange Language (GXL) is a collaborative effort from several academic and industrial research institutes to come up with a standard exchange format and a set of standard meta-models for information exchange between reengineering tools [34]. FAMIX is one of the main input meta-models that are taken as a basis for a standardized program entity level meta-model.

Instead of purpose-built reengineering environments, generic meta-data repositories can be used. Well-known examples are Unisys UREP and the Microsoft Repository [50]. These repositories attempt to address the general problem of sharing models between a large variety of different software tools. The advantages of these kinds of tools are that they are open to any information model and offer industry standard integration paths to existing systems, for instance, using XMI [39]. All functionality for storing, querying and exchanging information is available. The disadvantage is that they need considerable tailoring for specific uses and can be an overkill for the task at hand.

14. CONCLUSIONS

Based on the experience we gained while designing the FAMIX meta-model, building the Moose reengineering environment, and analyzing existing reengineering environments such as Datrix, Rigi, and SOUL we defined a design space based on a set of criteria that describe the underlying decisions that shape the design and functionality of the reengineering environment. We discussed requirement issues such as the level of detail, the way entities are referenced, and grouping facilities. Then we discussed implementation issues that have an impact on the functionality offered by a reengineering platform. In particular we discussed different approaches of interchange format, incremental loading of entities, entity reference schema and their dependencies.

We validated the design space by evaluating how different reengineering environments can be described in terms of the criteria presented. In future we would like to evaluate the available environments regarding all the axes we defined.

ACKNOWLEDGEMENTS

The authors would like to thank Serge Demeyer, Tim Lethbridge, Oscar Nierstrasz, Tamar Richner, Michele Lanza, and the anonymous reviewers for their comments on the early drafts of this article. We thank Roel Wuyts, SOUL's main developer, for information about the internal details of SOUL and Ric Holt for information relating to Datrix.

REFERENCES

1. Lehman MM, Belady L. *Program Evolution—Processes of Software Change*. Academic Press: London, 1985.
2. Bellay B, Gall H. A comparison of four reverse engineering tools. *Proceedings WCRE'97*. IEEE Computer Society Press: Los Alamitos CA, 1997; 2–11.
3. Armstrong MN, Trudeau C. Evaluating architectural extractors. *Proceedings WCRE'98*. IEEE Computer Society Press: Los Alamitos CA, 1998; 30–39.
4. Sim SE, Storey M-AD. A structured demonstration of program comprehension tools. *Proceedings WCRE 2000*. IEEE Computer Society Press: Los Alamitos CA, 2000; 184–193.
5. Lethbridge TC, Anquetil N. Architecture of a source code exploration tool: A software engineering case study. *Technical Report*, University of Ottawa, July 1997.
6. Laguë B, Leduc C, Le Bon A, Merlo E, Dagenais M. An analysis framework for understanding layered software architectures. *Proceedings of IWPC'98*. IEEE Computer Society Press: Los Alamitos CA, 1998.
7. Koschke R. Atomic architectural component recovery for program understanding and evolution. *PhD Thesis*, Universität Stuttgart, 2000.
8. Lane TG. A design space and design rules for user interface software architecture. *Technical Report*, Carnegie Mellon University, Software Engineering Institute, November 1990.
9. Wegner P. Dimensions of object-based language design. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87). ACM SIGPLAN Notices* 1987; **22**(12):168–182.
10. Finnigan P, Holt R, Kalas I, Kerr S, Kontogiannis K, Mueller H, Mylopoulos J, Perelgut S, Stanley M, Wong K. The software bookshelf. *IBM Systems Journal* 1997; **36**(4):564–593.
11. Bell Canada. DATRIX abstract semantic graph reference manual (version 1.4). *Technical Report*, Bell, Canada, May 2000.
12. Lapierre S, Lagué B, Leduc C. Datrix^{TM} source code model and its interchange format: Lessons learned and considerations for future work. *Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000)*, June 2000.
13. Tilley SR, Müller HA, Whitney MJ, Wong K. Domain-retargetable reverse engineering. *Proceedings CSM '93. The Conference on Software Maintenance*, September 1993. IEEE Computer Society Press: Los Alamitos CA, 1993; 142–151.
14. Wong K. The Rigi user's manual—version 5.4.4. *Technical Report*, University of Victoria, 1998.
15. Demeyer S, Tichelaar S, Ducasse S. FAMIX 2.1—the FAMOOS information exchange model. *Technical Report*, University of Berne, 2001.
16. Demeyer S, Ducasse S, Lanza M. A hybrid reverse engineering platform combining metrics and program visualization. *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*, Balmas F, Blaha M, Rugaber S (eds.). IEEE Computer Society Press: Los Alamitos CA, 1999.
17. Tichelaar S, Ducasse S, Demeyer S, Nierstrasz O. A meta-model for language-independent refactoring. *Proceedings ISPSE 2000*. IEEE Computer Society Press: Los Alamitos CA, 2000.
18. Demeyer S, Ducasse S, Nierstrasz O. Finding refactorings via change metrics. *Proceedings of the 2000 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'2000). ACM SIGPLAN Notices* 2000; **35**(10):166–178.
19. Lanza M, Ducasse S, Steiger L. Understanding software evolution using a flexible query engine. *Proceedings of the Workshop on Formal Foundations of Software Evolution*, 2001.
20. Ducasse S, Lanza M, Tichelaar S. Moose: An extensible language-independent environment for reengineering object-oriented systems. *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

21. Wuyts R. Declarative reasoning about the structure object-oriented systems. *Proceedings TOOLS USA '98 Conference*. IEEE Computer Society Press: Los Alamitos CA, 1998; 112–124.
22. Mens K, Wuyts R, D'Hondt T. Declaratively codifying software architectures using virtual software classifications. *Proceedings TOOLS-Europe 99*, June 1999. IEEE Computer Society Press: Los Alamitos CA, 1999; 33–45.
23. Mens K, Michiels I, Wuyts R, Supporting software development through declaratively codified programming patterns. *Proceedings SEKE 2001, International Conference on Software Engineering and Knowledge Engineering*, Buenos Aires, Argentina, 13–15 June. Knowledge Systems Institute, 2001; 236–243.
24. Linos PK, Schach SR. Comprehending multilanguage and multiparadigm software. *Proceedings Short Papers of ICSM'99*, August 1999. IEEE Computer Society Press: Los Alamitos CA, 1999; 25–28.
25. Tichelaar S. Modeling object-oriented software for reverse engineering and refactoring. *PhD Thesis*, University of Berne, December 2001.
26. Gallagher KB, Lyle JR. Using program slicing in software maintenance. *Transactions on Software Engineering* 1991; **17**(18):751–761.
27. Larsen L, Harrold MJ. Slicing object-oriented software. *Proceedings ICSE '96*. IEEE Computer Society Press: Los Alamitos CA, 1996; 495–505.
28. Wind River. *SNiFF+*. 2000.
29. Lanza M, Ducasse S. Understanding software evolution using a combination of software visualization and software metrics. *Proceedings of LMO 2002*. Hermes: Paris, 2002; 135–149.
30. Jazayeri M, Gall H, Riva C. Visualizing software release histories: The use of color and third dimension. *ICSM'99 Proceedings (International Conference on Software Maintenance)*. IEEE Computer Society Press: Los Alamitos CA, 1999.
31. Ducasse S, Demeyer S (eds.). *The FAMOOS Object-Oriented Reengineering Handbook*, University of Berne, October 1999. See http://www.iam.unibe.ch/~famoos/handbook.
32. Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam. *Object Constraint Language Specification (version 1.1)*. Rational Software Corporation, September 1997.
33. Object Management Group. Meta object facility (MOF) specification. *Technical Report ad/97-08-14*, Object Management Group, September 1997.
34. Holt RC, Winter A, Schürr A. GXL: Towards a standard exchange format. *Proceedings WCRE'00*. November 2000. IEEE Computer Society Press: Los Alamitos CA, 2000; 162–171.
35. CDIF Technical Committee. CDIF framework for modeling and extensibility. *Technical Report EIA/IS-107*, Electronic Industries Association, January 1994. http://www.cdif.org/.
36. World Wide Web Consortium. Resource Description Framework (RDF) model and syntax specification. *Technical Report*, World Wide Web Consortium, February 1999.
37. Saint-Denis G, Schauer R, Keller RK. Selecting a model interchange format. the SPOOL case study. *Proceedings Thirty-Third Annual Hawaii International Conference on System Sciences*. IEEE Computer Society Press: Los Alamitos CA, 2000.
38. Holt RC. An introduction to TA: The Tuple-Attribute language. *Technical Report*, University of Waterloo, November 1998.
39. Object Management Group. XML Metadata Interchange (XMI). *Technical Report ad/98-10-05*, Object Management Group, February 1998.
40. DeRose S, Maler E, Orchard D. XML Linking Language (XLink) version 1.0—w3c proposed recommendation 20. *Technical Report PR-xlink-20001220*, World Wide Web Consortium, December 2000.
41. Czeranski J, Eisenbarth T, Kienle HM, Koschke R, Plödereder E, Simon D, Zhang Y, Martin J-F, Würthner M. Data exchange in Bauhaus. *Proceedings WCRE'00, Exchange Formats Workshop*. IEEE Computer Society Press: Los Alamitos CA, 2000; 293–295.
42. Hassan A, Holt R, Lague B, Lapierre S, Leduc C. E/r schema for the datrix c/c++/java exchange format. *Proceedings WCRE'00, Working Conference on Reverse Engineering, Exchange Formats Workshop*. IEEE Computer Society Press: Los Alamitos CA, 2000; 284–286.
43. Ducasse S, Tichelaar S. FAMIX Smalltalk language plug-in. *Technical Report*, University of Berne, 2001.
44. Bär H. FAMIX C++ language plug-in 1.0. *Technical Report*, University of Berne, September 1999.
45. Nebbe R. FAMIX Ada language plug-in 2.2. *Technical Report*, University of Berne, August 1999.
46. Tichelaar S. FAMIX Java language plug-in 1.0. *Technical Report*, University of Berne, September 1999.
47. Object Management Group. Unified Modeling Language (version 1.3). *Technical Report*, Object Management Group, June 1999.
48. Demeyer S, Ducasse S, Tichelaar S. Why unified is not universal. UML shortcomings for coping with round-trip engineering. *Proceedings of UML'99 (The Second International Conference on The Unified Modeling Language)*, Kaiserslautern, Germany, October 1999 (*Lecture Notes in Computer Science*, vol. 1723), Rumpe B (ed.). Springer: Berlin, 1999.

49. Lethbridge TC. Requirements and proposal for a Software Information Exchange Format (SIEF) standard. *Technical Report*, University of Ottawa, November 1998. http://www.site.uottawa.ca/~tcl/papers/sief/standardProposalv1.html.
50. Bernstein PA, Bergsträsser T, Carlson J, Pal S, Sanders P, Shutt D. Microsoft repository version 2 and the open information model. *Information Systems* 1999; **24**(2):71–98.

AUTHORS' BIOGRAPHIES

Stéphane Ducasse obtained his PhD at the University of Nice-Sophia Antipolis and his habilitation at the University of Paris 6. He is Assistant Professor at the University of Berne. His fields of interests are: design of reflective systems, object-oriented languages design, composition of software components, design and implementation of applications, and reengineering of object-oriented applications. He is the main developer of the Moose reengineering environment. He loves programming in Smalltalk and is the Co-President of the European Smalltalk User Group. He has written several books in French and English: *La programmation: une approche fonctionnelle et recursive en Scheme* (Eyrolles 96), *Squeak* (Eyrolles 2001), *Object-Oriented Reengineering Patterns* (MKP 2002).

Sander Tichelaar received an MSc in Computer Science from the University of Gröningen, The Netherlands, in 1996 and a PhD in Computer Science from the University of Berne, Switzerland, in 2001. Currently he is working as a consultant for Daedalos Consulting in Zürich, Switzerland. His research interests include software evolution, tool support for program understanding and evolution and agile software development methodologies.