CDIF as the Interchange Format between Reengineering Tools

Oscar Nierstrasz, Sander Tichelaar and Serge Demeyer,
Software Composition Group, University of Berne, Switzerland,
{oscar,tichel,demeyer}@iam.unibe.ch

Abstract

Tool support is recognised as a key issue in the reengineering of large scale object-oriented systems. However, due to the heterogeneity in today's object-oriented programming languages, it is hard to reuse reengineering tools across legacy systems. This paper proposes a language independent exchange model, so that tools may perform their tasks independent of the underlying programming language. We have adopted CDIF[CDIF94a] as the basis for the exchange of information, using this model, between the reengineering tool prototypes in the FAMOOS project. The main reasons for adopting CDIF are, that firstly it is an industry standard, and secondly it has a standard plain text encoding which tackles the requirements of convenient querying and human readability. Next to that the CDIF framework supports the extensibility we need to define our model and language plug-ins.

The complete model is available at: http://www.iam.unibe.ch/~famoos/InfoExchFormat/
All comments are welcome: famoos@iam.unibe.ch.

1) Introduction

Reengineering object-oriented systems is a complex task. Therefore, many tools have been and are being developed to support developers in performing activities such as visualisation, metrics & heuristics and system reorganisation. However, legacy systems are written in different implementation languages (C++, Ada, Smalltalk and even Java). To avoid equipping all of the tools with parsing technology for all of the implementation languages, we have developed a model for information exchange between reengineering tools. The model is a language independent representation of object-oriented sources and should provide enough information for the reengineering tasks of the tools (see Figure 1).

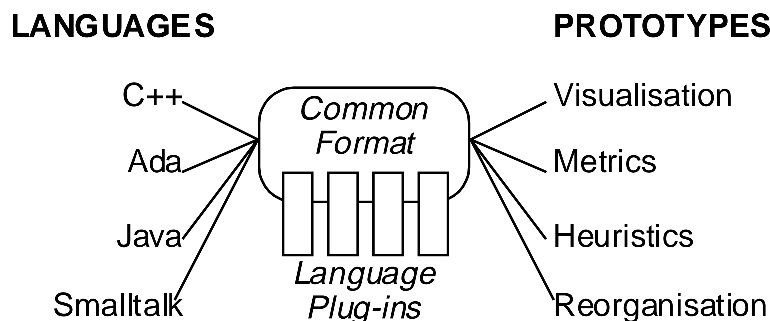

Figure 1: Conception of the Exchange Format

However, we cannot know in advance all information that is needed. Therefore, the model is extensible in a couple of ways. First, tools may need to work on the language specific issues

of a system. Therefore, we allow *language plug-ins* that extend the model with language-specific items, but don't break the language independent tools. Second, we allow *tool plug-ins* to extend the model so tools can, for instance, store analysis results or layout information for graphs, again without breaking other tools.

To exchange actual information, i.e. source code expressed in our model, we use the CDIF standard for information exchange[CDIF94a]. CDIF provides us with a standard way of exchanging models and model instances in ASCII representation.

Within the FAMOOS Esprit project we are currently using the exchange model together with its representation in CDIF in our reengineering tool prototypes. Prototypes written in different languages work on sources written in different languages. The first results look promising, but we need to do more experiments to validate the model and the gains it should provide in reusability and interoperability of our tools

2) Requirements Specification

Based on our experiences with the tool prototypes built so far, plus given a survey of the literature on reengineering repositories and code base management systems we specified the following requirements list. The list is split up in two, one part defining requirements concerning the data model, the other part specifying issues concerning the representation.

Data Model

1. *Extensible.*
   To handle the definition of language plug-ins, the data model must allow extensions with language specific entities and properties. Some tool prototypes may also need to define tool specific properties.
2. *Sufficient basis for metrics, heuristics, grouping and re-engineering operations.*
   To avoid a common denominator that would be ineffective for our goals, we set the lower limit for the model to everything that is required to experiment with the tool prototypes.
3. *Readily distillable from source code.*
   Since it is not our aim to define a model that covers all aspects of all languages, the upper limit to the information the model will contain, is what can be generated by basic code parsing (i.e. parsing without any interpretation of the obtained information, for instance, determining if a relation is an aggregation or a composition). The generated information should be usable by any tool , thus also by language independent tools.

Representation

1. *Easy to generate by available parsing technology.*
   Since we cannot wait for future developments, we must use parsers available today keeping an eye on short-term evolution. Within the FAMOOS project, parsing technology comes mainly from the FAST library part of the Audit platform. However, there are a number of other viable alternatives: like the SNiFF+ symbol table which is accessible via an API; like Ada compilers which provide standard API's for accessing internal data structures; like the tables generated by Audit which can be transformed in what is needed; like the Java inspection facilities part of Java.lang.reflect or even the Java byte code itself; like

Smalltalk inspection facilities and parsers that are part of every Smalltalk implementation.

2. *Simple to process.*
   As the exchange format will be fed into a wide variety of tool prototypes, the format itself should be quite easy to convert into the internal data structures of those prototypes. On top of that, processing by "standard" file utilities (i.e., grep, sed) and scripting languages (i.e., perl, python) must be easy since they may be necessary to cope with format mismatches.
3. *Convenient for querying.*
   A large portion of re-engineering is devoted to the search for information. The representation should be chosen so that it may easily be transformed into an input-stream for querying tools (i.e., spreadsheets and databases).
4. *Human readable.*
   The exchange format will be employed by (buggy) prototypes. To ease debugging, the format itself should be readable by humans. Especially, references between entities should be by name rather than by identifiers bearing no semantics.
5. *Allows combination with information from other sources.*
   Although most of the data model will be extracted from source code, we expect that other origins can provide input as well. Especially CASE tools with design diagrams (e.g., TDE or Rational/Rose) are likely candidates. Thus, the representation should allow merging information from other origins. Note that — just like with the "human readable" requirement— this implies that references between entities should be by name rather than by identifiers bearing no semantics.
6. *Supports industry standards.*
   Since the tool prototypes must be utilised within an industry context, they must integrate with whatever tools already in use. Ad hoc exchange formats (even when they can be translated with scripts) hinder such integration, and --when available-- the representation should favour an industry standard.

3) The Information Exchange Model

In this section we briefly describe the model for information exchange. A complete description can be found in [Deme98]. The inheritance structure of the complete model is shown in Figure 2. `Object`, `Property`, `Entity` and `Association` are made available to handle the extensibility requirement (see "Requirements Specification" - p.2). For specifying language plug-ins, it is allowed to define language specific classes and to add language specific attributes to existing `Objects`. Tool prototypes are more restricted in extending the model: they can define tool specific `Properties` for and can add attributes to existing `Objects`. They are, however, not allowed to extend the repertoire of entities and associations. The grey entities show the core model. The core consists of the main OO entities, namely `Class`, `Method`, `Attribute` and `InheritanceDefinition`. For reengineering we need the other two, the associations `Invocation` and `Access`. An `Invocation` represents the definition of a `Method` calling another `Method` and an `Access` represents a `Method` accessing an `Attribute`[1]. These abstractions are needed for reengineering tasks such as dependency

[1] Actually, the complete model is more general: an `Invocation` is about behavioural entities (such as methods and functions) calling other behavioural entities and an `Access` is about a behavioural entity accessing a structural entity (such as attributes and global variables).

analysis, metrics computation and reorganisation operations. Typical questions we need answers for are: “are entities strongly coupled?”, “which methods are never invoked?”, “I change this method. Where do I need to change the invocations on this method?”.

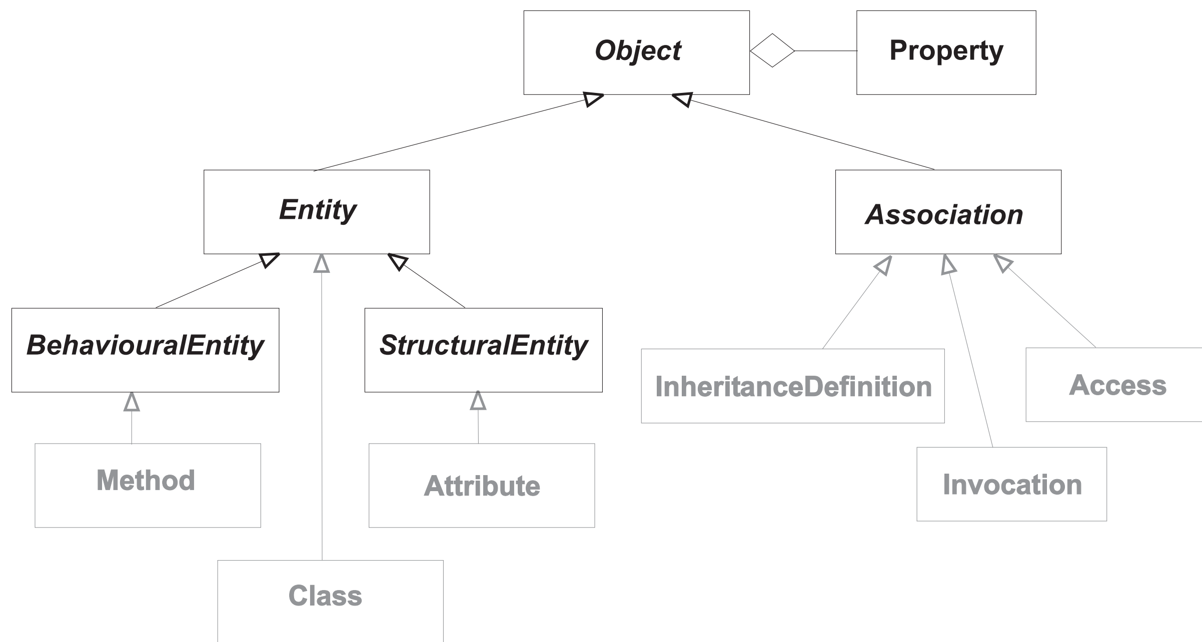

Figure 2: Basic structure of the complete model

4) CDIF as Interchange Format

To satisfy the requirements for information exchange between tools, we have chosen the CDIF standard [CDIF94a] as the basis for transferring information between tools. This choice at least satisfies the "supports industry standards" and the "extensible" requirements. Moreover, CDIF is open with respect to the specific format for a transfer, or —to state it in CDIF terminology— allows for different syntaxes and encodings. By adopting the CDIF syntax SYNTAX.1 with the plain text encoding ENCODING.1 (see [CDIF94b] and [CDIF94c]), we also satisfy the "human readable" and "simple to process" requirements.

CDIF has proven to be a proper solution for our purposes. However, the explicit definition of associations and the lack of multi-valued string attributes leads to verbose transfers that are difficult to read for humans and hinders the merging of information coming from different sources. Also, there are some things we found unclear while reading the CDIF specifications. Therefore, this part of the appendix describes our interpretation of the CDIF standard.

4.1 Avoid Explicit Relationships

We avoid explicit relationships. This might seem a bit strange at first, but our experiments have shown that heavy use of CDIF relationships compromises the "human readability" and "convenient for querying" requirements. First of all, information gets scattered around in the transfer instead of being nicely encapsulated in the entity it belongs to. This makes debugging difficult and makes the use of simple grep-like tools almost impossible. Next to that, CDIF relationships employ meaningless identifiers –unique within the transfer only– instead of

references by name. The latter also hinders the combination of information from different sources.

Below is an example of how we encapsulate a "belongsToClass" attribute in Method, instead of defining an explicit "Class.HasMethod.Method" relationship and instantiating it for every Class/Method association. Thus we get ...

```
(Class FM17
        (name "Widget")
        ...
)

(Method FM35
        (name "print")
        (belongsToClass "Widget")
        ...
)
```

instead of

```
(Class FM17
        (name "Widget")
        ...
)

...
(Method FM35
        (name "print")
        ...
)

...
(Class.HasMethod.Entity FM56 FM17 FM35)
```

4.2 Allow multi-valued String Attributes

To deal with many-to-1 relationships we need multi-valued string attributes. Indeed, we avoid explicit relationships to enhance the readability of a document and to ease combination of information from different sources. However, using a string attribute to encode a relationship (like we did above) only allows for 1-to-many relationships.

CDIF provides `IntegerList` and `PointList` in its set of basic data types, thus —in principle— CDIF permits the use of multi-valued attributes. Unfortunately, there is no basic data type that copes with multi-valued strings. Yet, the CDIF `TextValue` data type comes very near, thus in some rare occasions we interpret `TextValue` as a multi-valued string attribute.

In the original CDIF standard, a `TextValue` denotes a set of characters which is divided into blocks with a maximum of 1024 characters. The beginning of each block is marked by "#[" while the end is marked by "]#". The actual value of the text is the concatenation of the blocks.

To represent a multi-valued string attribute with a `TextValue`, we interpret each block in a `TextValue` as a separate string. Also, we require that each one of those strings must append a special delimiter character (which is "|") to its end so that the original multi-valued strings can be retrieved from the concatenated blocks. In the (unlikely) situation that a “|” appears in a string value it should be escaped with “\|”. Thus we get ...

```
(Invocation FM35
        (invokedBy "ScrollBar.print()")
        (invokes "print()")
        (candidates     #[Widget.print()|]#,
                        #[MotifWidget.print()|]#,
                        #[SwingWidget.print()|]#)
)
```

instead of (using CDIF relationships):

```
(Invocation FM35
        (invokedBy "ScrollBar.print()")
        (invokes "print()")
)

...

(Candidate FM45
        (value "Widget.print()")
)

(Candidate FM46
        (value "MotifWidget.print()")
)

(Candidate FM47
        (value "SwingWidget.print()")
)

...

(Invocation.HasCandidate.Candidate FM87 FM35 FM45)
(Invocation.HasCandidate.Candidate FM88 FM35 FM46)
(Invocation.HasCandidate.Candidate FM89 FM35 FM47)
```

Note that this is not a proposed solution for multi-valued strings. It is our workaround to keep to our requirements. We'd rather see a consistent use of lists in the CDIF standard, or at least support for a `StringList` as it is as useful as the other, already existing list types.

5) Why not UML?

The Unified Modelling Language (UML) [Booc96a] is rapidly becoming the standard modelling language for object-oriented software, even in industry. So, UML is a viable candidate for serving as the data model behind our exchange format. Nevertheless, UML is geared towards an analysis / design language and there exists no accurate and straightforward mapping from source code to UML. For instance, inheritance like applied in an implementation does not necessarily correspond to generalisation like specified in UML (e.g., in an implementation a Rectangle might be a subclass of Square while a correct generalisation is the other way around). Likewise, attribute definitions do not always correspond with aggregation (e.g., is a Rectangle an aggregation of two instances of Point or is it an aggregation of four integers). Thus choosing UML would violate the requirement that the data model should be readily distillable from source code (see p.2) and that's the first motivation to rule out UML.

Moreover, extracting an accurate UML model from source code is considered quite important for model capture. We will definitely investigate that topic in further depth, and we do not want to hamper such investigations by choosing a straightforward but inaccurate mapping. That is the second motivation to rule out UML.

Finally, UML does not include internal dependencies such as invocations and accesses. Those dependencies are necessary for problem detection and reorganisation operations. Thus, choosing UML would violate the requirement of being a sufficient basis for re-engineering operations (see p.2).

However, we relied heavily on UML in the terminology and naming conventions applied in our model to become independent of the implementation language. For example, we talk about attributes instead of members (C++) or instance variables (Smalltalk) and we talk about classes instead of types (Ada).

6) Why not the CDIF OOAD Core Subject Area?

The CDIF Object-oriented Analysis and Design Core Subject Area (CDIF OOAD) [CDIF96] is an (already two years old draft) subject area for representing and exchanging information found in existing object-oriented analysis and design techniques. The main reason not to take CDIF OOAD as our information model is that, just as it is the case with UML, it does not include the abstractions for invocations and accesses. Secondly, as our model is focussed on mapping source code rather than analysis and design, we don't need all the overhead for representing analysis and design abstractions such as, for instance, `DefinitionObject` or `OOADObject`.

7) Conclusion and open questions

This position paper reports on the use of CDIF for the exchange of information between reengineering tools. Several difficulties were encountered: our requirements demanded to minimise the use of CDIF relationships, so we introduced our own naming scheme parallel to the CDIF identifiers and we introduced our own notion of multi-valued string. Next to these technical problems, we had the "meta" problem that it was hard to start up using CDIF because of the unclearness of the documents. Take, for instance, the descriptions of the CDIF identifiers. The encoding document [CDIF94c] is less restrictive than the general framework description [CDIF94a] in its definition of the different CDIF identifiers. The general description is the only document that talks about the 32-character limit of the identifiers and uniqueness of names. Next to that, much of the knowledge how to use CDIF, especially for defining our exchange model in CDIF, had to be reverse engineered from the examples. An *annotated* example which contains of all typical parts (header, metamodel and model sections), including information on what links to what, would be very useful. In the current examples, identifiers from non-displayed subject areas are referred to, making it hard to determine what is going on. And why not using names instead of numbers for the meta identifiers[2]? This would increase understandibility a lot. We defined the FAMOOS Information Exchange Model in CDIF [Tich98], but we are still not 100% sure if the definition is syntactically right.

---

[2] A logical proposal would be to use the unique names of entities as their meta identifier. This is the way we did it in the CDIF definition of the presented exchange model [Tich98]. A problem occurs for unique names, typically the names of relations, that exceed the 32-character limit.

For the workshop we are interested in the following issues:

- Experiences of others who use CDIF as the base for the exchange of information. Do they have similar problems? Do they have other problems?
- We are in the early stages of deploying CDIF in our tool prototypes. Do people have success (or non-success) stories about the improved interoperability of their tools? Have people used CDIF for large scale projects? How does CDIF scale?
- Is their any tool support (e.g syntax checkers or support for generating CDIF definitions)?

Acknowledgements

This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT IV programme Project no. 21975 (FAMOOS, see http://www.iam.unibe.ch/~famoos/).

References

[Booc96a] Booch, G., Jacobson, I. and Rumbaugh, J, The Unified Modelling Language for Object-Oriented Development. See http://www.rational.com/.

[CDIF94a] CDIF Technical Committee, "CDIF Framework for Modeling and Extensibility", Electronic Industries Assocaiation, EIA/IS-107, January 1994. See http://www.cdif.org/.

[CDIF94b] CDIF Technical Committee, "CDIF Transfer Format Syntax SYNTAX.1", Electronic Industries Association, EIA/IS-109, January 1994. See http://www.cdif.org/.

[CDIF94c] CDIF Technical Committee, "CDIF Transfer Format Encoding ENCODING.1", Electronic Industries Association, EIA/IS-110, January 1994. See http://www.cdif.org/.

[CDIF96] CDIF Technical Committee, "CDIF - Integrated Meta-model - Object-oriented Analysis and Design Core Subject Area", Electronic Industries Association, EIA/IS-xxx, CDIF-DRAFT-OOAD-V01, January 1994. See http://www.cdif.org/.

[Deme98] Serge Demeyer and Sander Tichelaar, "Definition of the FAMOOS Information Exchange Model - Version 1.1", Technical Report, 1998. See http://www.iam.unibe.ch/~famoos/InfoExchFormat/.

[Tich98] Sander Tichelaar, "CDIF Definition of the FAMOOS Information Exchange Model - Version 1.1", Technical Report, 1998. See http://www.iam.unibe.ch/~famoos/InfoExchFormat/.