The Design and Implementation of a Metadata Repository

David Talby, Dotan Adler, Yair Kedem, Ori Nakar, Noa Danon, Arie Keren

MAMDAS, Israeli Air Force

Abstract. We present a tool which handles all the metadata created and used throughout a large-scale enterprise software project. The tool is used to visually create and edit different metadata specification documents, such as data entities, visual forms, queries, interfaces, transactions and so forth, and then automatically generate code from them. This enables a faster development process, as well as the design of a broad technical framework which fully separates the business logic code from technical concerns. The repository uses a unique unified data model and type system, allowing us to define and edit any level of metadata – from meta-metadata models to actual applicative data – using the same tool. This paper describes both the tool's design as well as the development process which it supports and improves.

INTRODUCTION

Having started a multi-year enterprise software project, employing tens of developers and supporting a variety of external interfaces, we set out to build its development environment. There are good tools and techniques for individual tasks, but – as in most large-scale systems – major integration overheads exist between the outputs of different stages of the development process. Consider the following issues.

After finalizing the system's requirements, a team of functional designers – experts in the business domain – begins writing detailed specifications of the system's entities, forms and transactions. Thousands of pages of documentation are created. They are transferred to the programmers, who read them and write code accordingly, and to the testing team, which re-reads them and prepares tests. Much of this work is highly repetitive – coding the names and types of an entity's fields, for example, and checking that required fields have indeed been filled at the right point. Yet, if someone has to read this fields list and manually translates it to code, then mistakes will happen – so tests must be written as well. Automating this passage can then potentially cut this type of work by two-thirds – not counting the reduced cost of training for coding and testing teams. The specification writers now have to be more formal and precise, and may take longer – but any "extra" formality required here only replaces a larger communication cost between teams later on, to clarify loose ends.

In parallel with writing detailed specifications, the technical teams begin designing an application framework. An object-oriented framework is a generic set of classes and interfaces, which specific applicative components inherit and extend [Lewis]. A well-written framework enhances reuse, hides low-level issues, and is easily extensible; well-known frameworks are MFC for client applications, and COM+ and EJB for server-side programs. It is nowadays common practice to build a customized framework on top of these, to meet a specific application's demands [Fayad et al]. This is usually done by deduction from requirements or early specifications. For example, from reading that a specific data structure must notify others when it changes, it may be decided to design a generic events or notification scheme. Such an approach often lacks a "big picture" view, and may lead to expensive overwork when the specifications rarely use a tough-to-implement feature, or decide to use a "minor variation" on the usual behavior which requires major technical changes.

A better practice is to establish a formal "contract" before beginning to write the detailed specifications: First agree on the framework, and fill the details afterwards. This way the functional experts team makes generic requirements, but then commits to use them only, or at least knows that changes will be expensive. The technical teams can know which generic mechanisms will be required, and design so that all cases are covered.

Having this done, the next obvious act would be to write the detailed design in a more formal way than free text. For example, if each field in a form has a "required" flag, a field type and so on, it is better to supply the specification writers with a form, in which the flag would be a checkbox and the type taken from a combo-box with a fixed list of options. This makes the field's definition faster, and enables direct code generation from this data. This also prevents the problematic "extra copy" of the data by programmers, and its costs – coding time, planning and executing tests, variations in coding style and level, and extended yet required training.

This paper describes the tool and process used to write, manage and use detailed specifications in our project. It is used for many formats of detailed specification documents – table 1 lists some of them. Such a repository of functional documents is called a Metadata Repository: a store of data about the system's data [Marco].

Kind	Description
First-Class Entities	Entities including their fields (with their properties), tables,
Second-Class Entities	Data objects which are embedded in entities
Field types	Properties of types shared by many fields and entities
Visual forms	Displayed fields, tables, buttons and toolbars
Form layouts	Screen coordinates, orderings, and customization options
Query definitions	Types of queries, and attributes which can be queried
Messages	Errors, warnings and other user messages
Menus	Menu actions, popup menus and combo-boxes
Interfaces	Fields and formats sent or received from other systems
Transactions	Parameters, entities involved, records locked, processing
Capabilities	Functional mechanisms that data entities can support

Table 1: Specification Document Kinds

FEATURES OF METADATA

As varied as they are, all documents have several common properties, on which a generic tool could be built.

First, they tend to have hierarchical structures – a visual form specification, for example, has global attributes, a list of fields (with their own properties), a list of actions (again, with properties), and a list of sub-forms (which contain fields, actions and sub-forms as well). Hence an XML-like data structure is appropriate, while a relational database is less natural [Bray et al].

Second, these documents change a lot during the normal course of a system's lifetime, making integrated configuration control a necessity [White]. This is especially true since different versions of a document are often used concurrently by different teams – while version *N* is being coded, version *N+1* is being designed, and version *N+2* specified.

Third, there are many different links between documents, which should be managed – for example, when a field's type in an entity changes, this may affect database tables in which it is stored, forms in which it's displayed, other systems to which it's sent, queries in which its value is examined, other entities holding a reference to it, and so forth. The ability to formalize such links, and query what has a link to a given document (in a given configuration control baseline), is highly important keeping the system consistent over time.

A fourth major trait of any major system is its tendency to change. In particular, the schemas themselves change as the framework evolves, and new schemas are created as the project enters new areas – interfaces to other systems, maps and geographic data, and so on. To support this, the metadata repository must enable the creation and editing of document kinds as well as documents – that is, "kinds" are specification documents by themselves. There's even a document that specifies the kind "kind", just like the specification of kind "entity".

At a lower lever of metadata, another requirement is the ability to edit actual data entities that the system is supplied with. For example, in a banking system, there will be entities such as "Bank" and "Currency", and forms to edit such entities, but the system will also be supplied with a database of existing bank and currencies. These must be specified as well – including visual editing, queries and configuration control. That is, each predefined bank will be a document, whose "kind" will be the "bank entity" document.

These documents vary greatly in their properties, hierarchy and use. No special-purpose language, such as UML [Booch et al, OMG-UML] or XML Schema [Biron et al, Malhotra, Thompson] was sufficiently strong or convenient to support all kinds. Yet, the specification writers had to be supplied with a simple and unified user interface to edit data – so to enable such an interface, a simple and unified data model had to be designed.

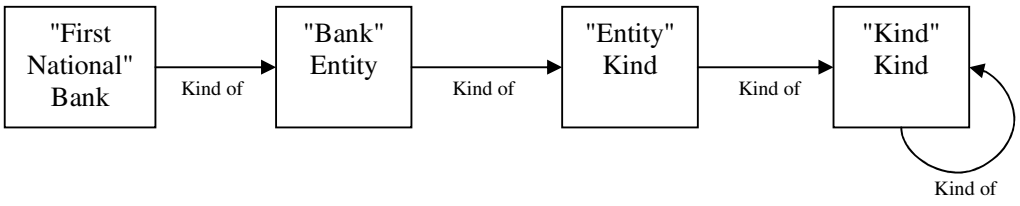

Figure 1. Kind-Of Relations

Consider the above example of the specifications of a particular bank, based on the bank entity as its kind. As a reminder, the “bank entity” document itself has a document defining its kind – “entity kind” – which by itself is defined by the “kind kind” document. The “kind kind” document is defined by itself, closing the chain. Figure 1 visualizes these relationships.

Albeit the initial similarities, this model varies greatly from the common model of data and metadata. For example, the OMG MOF model [Kumaran, OMG-MOF] and XML metadata interchange (XMI) [OMG-XMI] includes four layers mirroring the data described in Figure 1. The standard's purpose is to describe how metadata such as UML diagrams should be written in XML, in order to make exchanging them between programs easier.

The bottom layer includes "actual" data – instances of the application's classes, such as specific banks and currencies. The models layer defines the schemas or interfaces of the data; it can be written in UML [OMG-UML] or IDL [OMG-IDL], and the XMI standard describes how it should be translated to XML. XML Schema editors can also be used in this layer, although as with UML the output must be transformed to the XMI format. The meta-models layer defines the schemas of the schemas: for example, how UML classes and packages look like, or how IDL elements are composed. The top, meta-meta-model layer, is hardwired, and defined how to describe the meta-models.

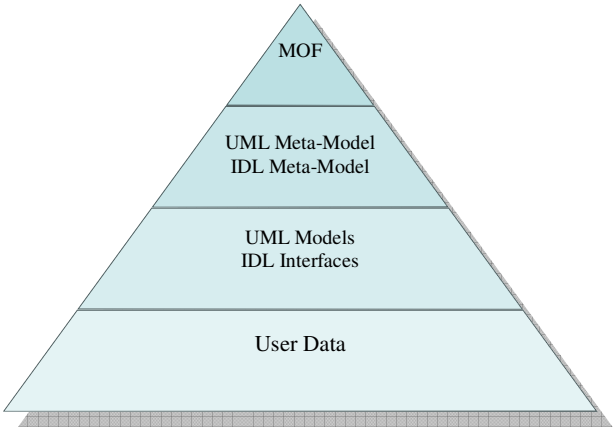

Figure 2. MOF Metadata Layers

This division of layers is insufficient for our needs, for several reasons:

- The metadata layers and the bottom data layer (used to specify predefined constants) have different formats and different semantics, and therefore cannot be created and edited by the same tool, or validated by the same parser.
- UML and IDL are limited to defining only certain kinds of metadata, such as classes, packages, methods and fields. Since they are standards, it is impossible to extend them or to define other application-specific kinds. In other words, the meta-model level is "read-only", which is unacceptable.
- The top layer is hard-wired. This is an unnecessary handicap; for example, our project defines that each document must have a unique Hebrew name, and optionally an icon. Both of these may well be undesired in other projects in which the repository will be used.
- In the MOF model, a document's kind (UML, IDL or Schema) is fixed once it is created, since it defines the document's data structure and since the metadata layers are fixed. Our data model removes this restriction; the practical benefits are detailed in the next page.

Consequently, we have designed a unified data model, in which all data has the same format, the same access to the metadata defining it (its kind), and the ability to be used at any layer of metadata and even define new layers. It is based on these three principles:

1. All data is stored in documents.
2. Every document has a kind document.
3. Every document that defines a list of fields may be used as a kind.

For example, the document on the right is the kind of the document on the left:

```
<Document>
    <FullName>
        <First>John</First>
        <Last>Doe</Last>
    </FullName>
    <Age>26</Age>
</Document>
```

```
<Document>
    <Name>Person</Name>
    <Fields>
        <Document>
            <Name>FullName</Name>
            <Type>FullName</Type>
            <IsRequired>true</IsRequired>
            <IsUnique>true</IsUnique>
        </Document>
        <Document>
            <Name>Age</Name>
            <Type>NaturalNumber</Type>
            <IsRequired>false</IsRequired>
        </Document>
    </Fields>
</Document>
```

Figure 3. Document XML and its Kind Document XML

Note that it doesn't matter what the document on the left is, or what its level of metadata is. It can be an entity, since entities have fields; it can be a form, an interface, or a table; it can also be any "kind", defining a specific entity, form or table; it can even be "kind kind", defining what entities or forms are. Instead of supporting multiple metadata languages, the "kind-of" relation is now a closed relation, which may exist between (almost) any two members (i.e. documents) of the repository. There is then no limit to adding more levels of metadata – the only measure of depth is the length of the longest route in the "kind-of" relation graph.

The relation between a document and its kind is not "hardwired" – in fact, it is not written anywhere inside it. This is highly contradictory to other schema standards, and easily enables changing kinds easily. This means that in order to open and edit a document, a tool must be supplied with two files – the document itself and its kind. We have imposed a convention, by which the document's file name is combined by both its name and its kind's name, so all information is available when opening a file.

The ability to change a document's schema after it's created may seem like a theoretical peculiarity at first, but it is actually used routinely in our work. There are at least two scenarios in which the same document may have to be viewed using different "kinds" during its life:

- Copying a document of one kind to another kind. For example, a common use case in our development process is to define an entity (including its fields, tables and their properties), and then create a form based on it. It is required that the "create form based on entity" operation will copy all fields, including the entire hierarchy, and all shared properties. Instead of coding this specific operation, which will have to be maintained each time one of the two kinds changes; an entity can simply be copied as-is and be named as a form of the same name. The new "form" will open correctly, hierarchy included. This provides a generic, zero-code copy and move operations: just rename the file from "bank entity" to "bank form", double-click it, and it will open as a form.
- Transferring documents between users working under different configuration control views. For example, a specifications writer may need to send a patch of an entity to the programming team. The programmers are now coding version N of the system using the version N "entity kind", while the spec writers are using a different N+1 version of the kind. Still, no conversion is necessary. This transfer may happen at all layers of metadata.

Another implication of this capability is that the "kind-of" relation is not the same as the "schema-of" relation. While both kinds describe XML element names and some constraints on these elements (type, required and unique fields, minimum and maximum value and so forth), a schema requires that only the defined fields will appear in the validated document. In contrast, if a document is viewed using a kind that does not "know" all its fields, the document is still legal, but these fields are simply ignored. This way, when an entity is copied to be a form, entity-specific fields, which are not defined for forms, will simply be ignored when the document is viewed as a form. They will be deleted when the document is first saved as a form, preventing the accumulation of irrelevant fields in documents.

REPOSITORY COMPONENTS

The metadata repository is composed of four software components, related as shown in Figure 5.

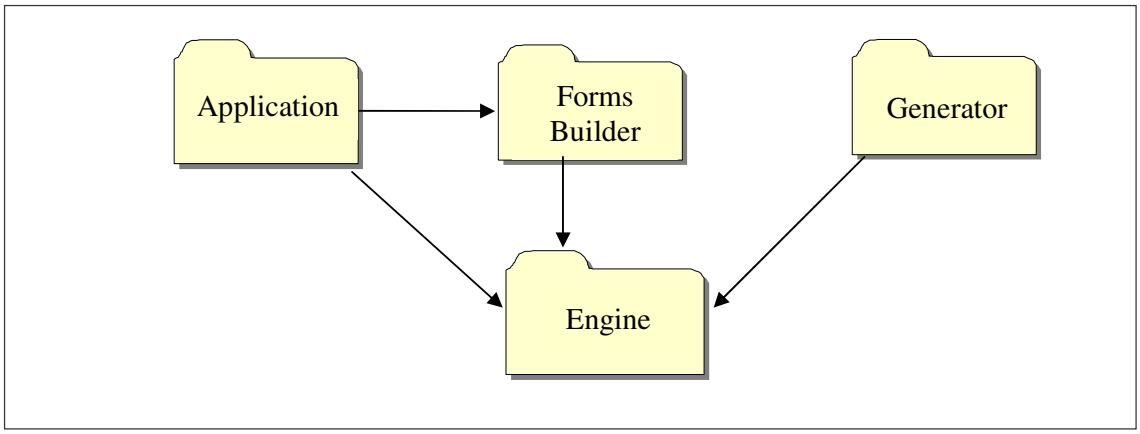

Figure 4. Repository Software Components

Engine. While the repository's files can be directly edited as XML, as some of our technically oriented users do, this is inadequate for most tasks. Visually presenting or editing a document, generating code, or sending data to other programs requires a view of the data that includes viewing the data with the correct version of its kind, merging inherited and overridden values, adding capabilities, and loading the types of all fields, which may use inheritance and references to other types and capabilities as well.

To prevent each client of the data from having to deal with all this, the engine is a DLL which provides a COM object model [Microsoft-COM, Williams] of the repository and all its data. It defines classes such as *Document*, *Table* and *Definition*, which completely encapsulate all data access and manipulation. For example, a loaded *Document* object will have the correct fields of the current version of its type; the values of its fields will be available, whether they were inherited or locally defined; and methods such as *save* and *validate* can be applied on it. Users of the engine are completely exempted from having to directly access the file system or parse XML.

The availability of the engine as a set of COM interfaces allows us to integrate the repository easily with the rest of the development environment: the data can now be easily accessed from products such as Microsoft Word (for reports), Rational Rose (for UML views), ClearCase (configuration control), Mercury TestRunner (automatic testing), Internet Explorer and others. This is a key factor of the repository's ability to serve many different users, and be a major communication and integration tool between the project's teams.

Another use of the engine is to maintain a cache of created documents, thereby minimizing the amount of XML parsing and analysis to do, as well as an index of names and physical locations of documents of all kinds. In contrast to common database engines, the index must be private to each user – because of configuration control, different users may concurrently be using different views and versions of the repository's data.

Forms Builder. The forms builder receives a *Document* object that the engine created, and creates a visual form that can display and edit a document. Like documents, forms are hierarchical and navigated using a tree display. The forms are intended to be used by non-technical people, and to help the editing process. Since kinds can be created and edited dynamically, forms must also be created dynamically, based on the document's kind. This is the main difficulty in implementing the forms builder, but it also makes it extremely flexible.

The form is built based on the document kind's field types. A Boolean field becomes a check box, an enumeration becomes a combo box, and a reference becomes a combo box of documents of the referred kind, and is also a hyperlink once a document is chosen. Records are forms which appear under the main form in the document's tree view. Collection become editable grids, whose columns are again check boxes, spin edits or combo boxes reflecting the column's type. Other properties such as the input mask and maximal length of text and numeric fields are also applied, and required or unique fields are colour-marked to indicate this.

By default, fields are put on a form at constant space from each other. This layout scheme is simple, automatic (which it must be), but not always the most aesthetic. This can be handled by using personalization: each user can customize the layout of each form; layouts are saved as documents (of kind "layout", naturally), and can be published to others. Since forms and layouts can be built for any level of metadata, they can also be used for building the forms of the application itself, not just the development environment, and this is indeed the way we design and use form layouts in our project. The same code can be used to enable end-user personalization of form layouts in our application, making this complex feature very simple to support.

Application. The visual editor of documents is a simple Windows application, which enables the creation and editing of documents of all kinds through visual forms. This is the simplest component – it uses the engine to perform all queries and data manipulation, and the forms builder to actually display the hierarchical forms. The application is the main everyday work tool of the specification writers.

Generator. Once the repository contains specs and the technical framework exists, framework-specific code must be generated to build the working application from the specs. The generated files may include source code, SQL, XML Schemas, server configuration files, documentation in various formats and so on. The generator is a tool which enables the editing of templates, which define what should be generated from documents. For example, Figure 6 defines a C++ class of a data entity, and private data members for all its fields. In the first line, the entity's name – as read from its document – its used as the class name; afterwards, the generator loops over its fields table, and for each field uses its name and its type's name to generate the private data member.

The template engine is generic – the field names are not hard-coded, but dynamically read from the generated document's kind. Note that a template is validated against a given kind, and can then be used for all documents of that kind – for example, to generate source files for all data entities, all types and so forth.

```
class [Name] {
private:
[for Fields]
    [Type.Name] _[Name];
[/for]
}
```

Figure 5. Generator Sample Template

A FASTER DEVELOPMENT PROCESS

The unified data model and the unified type system provide a flexible basis, which the software components implement. We now turn back to the big picture, to examine the project's development process as a whole and see how the repository solves our initial problems.

Before beginning to write specifications or design the technical framework, the technical and functional teams must meet and agree on the exact structure of the repository's kinds. This includes which kinds exists – for example, are tables specified or do they have a standard user interface? – As well as which properties must be defined for each kind. This requires considerable forethought from the business domain experts, since they must decide exactly which areas of the application can be customized and which cannot. For example, assume that the application must support drag-and-drop behaviour of entities between tables; is this a special case whose behaviour must be specified, or is this regarded exactly as a delete operation on the first table and an insert operation on the second one? Each case requires the document "kind form" to contain different information.

This is not an easy task, which requires the business domain experts to think about the nuts-and-bolts of every aspect of the application. The output of this process includes not only the document kinds, but naturally also the principles of the system's user interface, access to transactions and to the database, interfaces to other systems, and so on. Whether a metadata repository is used or not, we highly recommend applying this process, since it is a precondition to designing a usable framework. The technical teams must know which parts of the application are common principles – and are then handled inside the framework – and which parts require hooks to define customized behaviour. The need to define document kinds before documents is a technical requirement that ensures this process happens.

Once the document kinds are in place, the business domain and technical teams can progress in parallel. At the point where a first version of the framework is ready, the data from the repository must be used to code the application-specific behaviour, and create a running application. At first this is done by manually handcrafting the code, but once it runs it is translated into templates. The generator is used from then on to create the application-specific code directly from the metadata as entered by the specification writers – "the fast route". The mass amount of work usually found in information systems projects – coding and testing entities, tables and forms – now becomes automated, allowing a much faster pace.

As the project continues, the framework and templates evolve – since kinds can be edited, and since existing documents are still valid even when their kind has changed, it is relatively easy to introduce new features. For example, to add the ability to send an entity to another system, the kind "entity" can be extended to declare which fields are sent when the entity is sent. This new per-field flag can be configured to be false by default, and from this moment on all existing entities will have this flag, with that default. Thanks to the dynamic binding between documents and their kinds, when the entity's form will be opened, its fields table will include the flag.

A HIGHER QUALITY DEVELOPMENT PROCESS

Following the universal Pareto principle, about 80% of the specifications can be automatically generated, while about 20% include customized business logic which must still be coded manually – "the slow route". This means that most business logic is simple enough to be either handled within the framework or generated to code, which besides speed gives us significant quality benefits: We have less code, which is more heavily reused. Naturally, such code also tends to be better tested and optimized, since problems in it show earlier and have a wider effect.

This technique of building enterprise-scale applications – by designing frameworks and meta-metadata models first, and generating code from them once detailed specs exist – is called model-based code generation [Sarkar]. It has several advantages other than speed, by reducing several widespread types of hidden costs:

- **Division of Required Knowledge**. Application developers needn't learn how to produce code to handle low-level technical aspects. The management of entities, queries, transactions, forms, errors and so on is hidden to them, and they only need to learn the interfaces supplied to use them and the hooks by which they add customized business logic. They should be experts in the business domain. On the other hand, the developers of the frameworks and generator templates require intimate knowledge of the chosen database, application server, web server, protocols and generic functional mechanisms, but relatively little knowledge of the business processes the application supports. This enables developers of both kinds to focus on becoming experts on fewer subjects.

- **Reduced Training**. A direct result of the division of required knowledge is that new people must learn less to start working in the project, in any team. Replacing personnel becomes faster and easier.

- **Design Coherence.** Model-based generation requires a very clear separation between the framework code and the business logic code. The customized business logic doesn't get entangled in lower-level code as often happens. Changing the framework or the templates can be done in one place, and the effect will ripple. Changing business logic cannot harm technical mechanisms.

- **Enforcement of Standards.** All developers must conform to both code and design standards, and this is enforced by "hard" technical means. The framework must provide clear interfaces to it, which by design requires separating interfaces and implementation. Clients of the framework (i.e. the business logic code) must use the naming conventions and design patterns imposed by the framework, in order to communicate with it correctly. Common design violations such as bypassing logical layers, breaking encapsulation and others – caused by ad-hoc concerns during rush-hour periods or by unwary developers – are much harder to induce.

- **Platform Independence.** Replacing the project's operating system, database, application server, web server, communication protocols, user interface components, mapping tools, management tool, logging tool or any other infrastructure can be done without rewriting business logic code. The change is done in one place – the implementation of the framework's interfaces of the replaced subject. The only tie is to the business logic's programming language –caused only by the "slow route" part of business logic.

A final explanation is warranted about the role of Object-oriented analysis and design and UML in the process. It may seem left out, but it's actually heavily used, as it was meant to: to convert use cases into sequence diagrams and class diagrams, for the detailed design of the technical framework, and to describe the relations between entities. However, the use cases are detailed in a generic manner, in the stage when the kinds are created – there are use cases for creating an entity, updating it and so forth. There is no need to repeat the analysis for each entity and form specified from then on. As for the mass of specified metadata, while some of it fits into the UML language (particularly relations between entities and types), most of it does not – the major portion of documents contains application-specific properties. The purpose of the metadata repository is to complement OOAD, by using it to design a reusable framework and template set once, and then reusing it to derive a complete working application from a repository of growing specifications.

SUMMARY

The three major expected rewards from the repository are focusing the functional teams on meta-requirements first, enabling the design of a stable technical framework; a formal representation of meta-data, enabling automatic code generation from much of it, saving coding and testing time; and a better user interface for writing meta-data and functional specifications in general.

We considered several commercial tools titled as metadata repositories [Bernstein, Platinum], but faced limitations such as strong coupling with other tools, weak configuration control, overall complexity, and an emphasis on integrating into existing rather than new projects. However, building the repository in-house gave

us distinct benefits, which makes this choice even nicer. First, we were able to use the entire Dynamic Forms component for the "real" system as well. Second, the functional designers work with the actual user interface of the final application – an extreme approach to providing users with early prototypes. Third, we built the type system from scratch by our own demands, which kept it relatively simple. And fourth, we were able to integrate the repository into our development environment – Rose, Word, ClearCase – lowering the total number of tools and procedures in use.

From the academic point of view, the unified data model as well as the unified type system (part of which is beyond the scope of this paper) are both new, with respect to current XML metadata standards and repositories. In contrast with the many academic projects in the area today, every feature of the models was backed by a real business need of a real-world, real-scale project. We therefore believe that the ideas used here deserve a higher academic focus in future research.

The repository is now deployed throughout the project, and in parallel still being developed and expanded. It is the primary communication tool between the projects' teams, combining knowledge of specifications, interfaces, tests and technical infrastructure, and is well worth its initial investment.

ACKNOWLEDGEMENTS

A large number of people has helped and contributed to the repository. Among those, we would like to specifically thank Yariv Lifchuk, Roy Itach, Guy Grinwald, Leon Marom, Yitzik Hershko, Yaron Telem, Uri Landau and Rami Marelly.

REFERENCES

Bernstein, Philip et al, "The Microsoft Repository", *ACM SIGMOD conference on very large databases (VLDB)*, 1997. See also http://www.acm.org/sigmod/vldb/conf/1997/P003.PDF

Biron, Paul and Malhotra, Ashok, eds., "XML Schema Part 2: Datatypes", See http://www.w3.org/TR/2000/CR-xmlschema-2-20001024/datatypes.html

Booch, Rumbaugh, and Jacobson, *The Unified Modeling Language User Guide*. Reading Mass.: Addison-Wesley, 1999. UML spec found at OMG home http://www.omg.org

Bray, Tim et el, "Extensible Markup Language (XML) 1.0 (2nd Edition)". World Wide Web Consortium, 2000. See http://www.w3.org/TR/REC-xml.

Bray, Tim et al, "Namespaces in XML", W3C, 1998. See http://www.w3.org/TR/REC-xml-names/

Fayad, Mohamed, Douglas Schmidt and Ralph Johnson, *Building Application Frameworks*, Wiley 1999.

Fayad, Mohamed, Douglas Schmidt and Ralph Schmidt, *Implementing Application Frameworks*, Wiley '99.

Kumaran, Ilango, "The Specification of the Metadata Interchange Format", See http://www.trcinc.com/knowledge/articles/Specification_For_The_Metadata_Interchange_TRCInc.pdf

Lewis, Ted. *Object Oriented Application Frameworks*. Manning Publications Co., Greenwich, CT, 1995.

Marco, David, *Building and Managing the Meta Data Repository*, Wiley, 2000.

Microsoft Corporation, "Microsoft COM Technologies", See www.microsoft.com/com

Object Management Group (OMG), "Interface Definition Language Specifications", See http://www.omg.org/gettingstarted/omg_idl.html"

Object Management Group (OMG), "Meta Object Facility (MOF) Specification Version 1.3", March 2000, See http://www.omg.org/docs/formal/00-04-03.pdf

Object Management Group (OMG), "Unified Modeling Language Specification", See www.omg.org/uml

Object Management Group (OMG), "XML Metadata Interchange Specification", See http://www.omg.org/technology/documents/format/xmi.htm

Platinum Corporation (CA), "Platinum Repository Product Information", See http://www.platinum.com/products/dataw/repos_ps.htm

Malhotra, Ashok and Maloney, Murray, ed., "XML Schema Requirements", W3C, 15 February 1999. See http://www.w3.org/TR/NOTE-xml-schema-req

Sarkar, Soumen and Cleaveland, Craig, "Code Generation Using XML Based Document Transformation". *TheServerSide.com*, See www.theserverside.com/resources/articles/XMLCodeGen/xmltransform.pdf

Thompson, Henry; Beech, David; Maloney, Murray and Mendelsohn, Noah, eds., "XML Schema Part 1: Structures", W3C, 2 May 2001, See http://www.w3.org/TR/xmlschema-1/

White, Brian, *Software Configuration Management Strategies and Rational ClearCase*, Addison-Wesley.

Willliams, Sara and Kindel, Charlie, "The Component Object Model: A Technical Overview", *Dr. Dobb's Journal*, December 1994.

BIOGRAPHY

David Talby is a software engineer at the MAMDAS operational software development unit of the IAF. He graduated M.Sc. in Computer Science (1999), MBA in Business Administration (1999) and B.Sc. in Computer Science (1997) from the Hebrew University.

Dotan Adler is working as a Software Architect at the MAMDAS software development unit in Israeli Air Force. He graduated B.Sc. in Computer Science from Tel-Aviv University (1995).

Yair Kedem is a senior system analyst at the MAMDAS software development unit in Israeli Air Force. He graduated B.Sc. in Computer Science from Bar-Ilan University in Ramat-Gan (1994) and M.A. in Business Management from Ben-Gurion University (2000).

Ori Nakar is a software engineer at the MAMDAS software development unit of the IAF, since completing the IDF's Mamram training course in 1999.

Noa Danon is a software engineer at the MAMDAS software development unit of the IAF, since completing the IDF's Mamram training course in 2000.

Arie Keren is a senior system engineer at the MAMDAS software development unit in Israeli Air Force. He graduated Ph.D. in Computer Science from Hebrew University in Jerusalem (1998), M.Sc. in Computer Science from Tel-Aviv Univesity (1991) and B.Sc. in Computer Engineering from Technion, Haifa (1983).